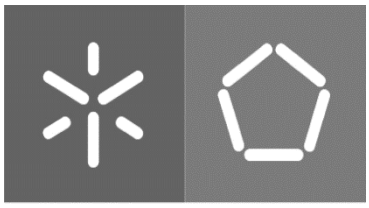
Universidade do Minho
Escola de Engenharia

José Pedro Antunes Ribeiro

A TrustZone-assisted Hypervisor Supporting Dynamic Partial Reconfiguration

Outubro de 2018



Universidade do Minho
Escola de Engenharia

José Pedro Antunes Ribeiro

A TrustZone-assisted Hypervisor Supporting Dynamic Partial Reconfiguration

Dissertação de Mestrado em Engenharia Eletrónica Industrial e Computadores

Trabalho efectuado sob a orientação do
Doutor Sandro Pinto

Outubro de 2018

Acknowledgements

Firstly, I would like to thank my advisor Dr. Sandro Pinto, for being always present throughout all of the development of my thesis. Thanks for proposing me a challenging project for my dissertation, while also enlightening me with ideas on the way to go whenever I encountered a dead-end. I would also thank you, for the motivation to push my limits and the opportunity of writing my the first scientific contribution.

I would also like to thanks Dr. Adriano Tavares, for the contribution he made in the last 2 years where he always pushed the embedded systems class limits, in ways that allowed us to grow as humans but also as the engineers of the future.

Following in the line, I want to give my thanks to José Martins and João Alves, the developers of µRTZVisor, and how they made my job so much easier, always being available to explain their reasons behind the implementation and full support when my doubts and problems using the hypervisor appeared.

To my fellow lab friends, Ângelo Ribeiro, Andersen Bond, Fransciso Petrucci, Hugo Araujo, José Silva, Nuno Silva, Pedro Machado, Sérgio Pereira and Ricardo Roriz, thanks for providing a great environment to work and the great collaborative spirit when there was the need, but also for the great friendship we made.

Finally, I could not let my family out of this note, specially my grandparents and godparents, as they were always there to support me and had the patience when the times were more complicated. To my parents, who have my "maior obrigado" for supporting me in my 17 years as a student, and little brother, who was the one always bringing a smile to my face on the roughest moments, I sincerely hope that I made you feel proud.

Abstract

Traditionally, embedded systems were dedicated single-purpose systems characterised by hardware resource constraints and real-time requirements. However, with the growing computing abilities and resources on general purpose platforms, systems that were formerly divided to provide different functions are now merging into one System on Chip. One of the solutions that allows the coexistence of heterogeneous environments on the same hardware platform is virtualization technology, usually in the form of an hypervisor that manage different instances of OSes and arbitrate their execution and resource usage, according to the chosen policy.

ARM TrustZone has been one of the technologies used to implement a virtualization solution with low overhead and low footprint. μRTZVisor a TrustZone-assisted hypervisor with a microkernel-like architecture - is a bare-metal embedded hypervisor that relies on TrustZone hardware to provide the foundation to implement strong spatial and temporal isolation between multiple guest OSes.

The use of Partial Reconfiguration allows the designer to define partial reconfigurable regions in the FPGA and reconfigure them during runtime. This allows the system to have its functionalities changed during runtime using Dynamic Partial Reconfiguration (DPR), without needing to reconfigure all the FPGA. This is a major advantage, as it decreases the configuration overhead since partial bitstreams are smaller than full bitstreams and the reconfiguration time is shorter. Another advantage is reducing the need for larger logic areas and consequently reducing their power consumption.

Therefore, a hypervisor that supports DPR brings benefits to the system. Aside from better FPGA resources usage, another improvement that it brings, is when critical hardware modules misbehave and the hardware module can be replaced. It also enables the controlling and changing of hardware accelerators dynamically, which can be used to meet the guest OSes requests for hardware resources as the need appears. The propose of this thesis is extending the μRTZVisor to have a DPR mechanism.

Resumo

Tradicionalmente, os sistemas embebidos eram sistemas dedicados a uma única tarefa e apenas limitados pelos seus requisitos de tempo real e de hardware. Contudo, como as plataformas de uso geral têm cada vez mais recursos e capacidade de processamento, muitos dos sistemas que executavam separadamente, passaram a apenas um sistema em plataforma recorrendo à tecnologia de virtualização, normalmente como um hipervisor que é capaz de gerir múltiplos sistemas operativos arbitrando a sua execução e acesso aos recursos da plataforma de acordo com uma politica predefinida.

A tecnologia TrustZone da ARM tem sido uma das soluções implementadas sem ter grande impacto na performance dos sistemas operativos. μRTZVisor é um dos hipervisores baseados na TrustZone para implementar um isolamento espacial e temporal entre múltiplos sistemas operativos, sendo que defere de outras uma vez que é de arquitectura microkernel.

O uso de Reconfiguração Parcial Dinâmica (RPD) permite ao designer definir várias regiões reconfiguráveis no FPGA que podem ser dinamicamente reconfiguradas durante o período de execução. Esta é uma grande vantagem, porque reduz os tempos de reconfiguração de módulos reconfiguráveis uma vez que os seus bitstreams são mais pequenos que bitstreams para a plataforma toda. A tecnologia também permite que nos FPGAs não sejam necessárias áreas lógicas tão grandes, o que também reduz o consumo de energia da plataforma.

Um hipervisor que suporte RPD traz grandes benefícios para o sistema, nomeadamente melhor uso dos recursos de FPGA, implementação de aceleradores em hardware dinamicamente reconfiguráveis, e tratamento de falhas no hardware. Se houverem módulos que estejam a demonstrar comportamentos inesperados estes podem ser reconfigurados. O uso de aceleradores reconfiguráveis permite que o hardware seja adaptável conforme a necessidade destes pelos diferentes sistemas operativos. A proposta desta dissertação é então estender o μRTZVisor para ter a capacidade de usar módulos reconfiguráveis por RPD.

Contents

List of Figures

List of Tables

List of Listings

Glossary

µRTZVisor	Microkernel Realtime TrustZone-assisted Hypervisor
AMBA	Advanced Microcontroller Bus Architecture
AP SoC	All Programmable System-On-Chip
API	Application Programming Interface
APU	Application Processing Unit
ASIC	Application Specific Integrated Circuit
AXI	Advanced eXtensible Interface
AXI DMA	Advanced eXtensible Interface Direct Memory Access
CPU	Central Processing Unit
DevCfg	Device Configuration Interface
DMA	Direct Memory Access
DMAC	Direct Memory Access Controller
DPR	Dynamic Partial Reconfiguration
DRAM	Dynamic Random Access Memory
DSP	Digital Signal Processing
FIFO	First-In First-Out
FIQ	Fast Interrupt Request
FPGA	Field Programmable Gate Array
FRI	FPGA Reconfiguration Interface
GIC	Generic Interrupt Controller
GPOS	General Purpose Operating System
hwMMU	hardware Memory Management Unit
ICAP	Internal Configuration Access Port
IoT	Internet of Things
IP	Intellectual Property
IPC	Inter-Partition Communication
IRQ	Interrupt Request
ISA	Instruction Set Architecture
MEMIF	Memory Interface

MIO	Multiplexed Input/Output
MM2S	Memory-Mapped to Stream
MMU	Memory Management Unit
OS	Operating System
OSFSM	OS Synchronization Finite State Machine
OSIF	Operating System Interface Module
PCAP	Processor Configuration Access Port
PCB	Partition Control Block
PL	Programmable Logic
PR	Partial Reconfiguration
PRR	Partial Reconfigurable Region
PS	Processing System
RAM	Random Access Memory
RM	Reconfigurable Modules
ROM	Read-Only Memory
RP	Reconfigurable Partition
RPC	Remote Procedure Calls
RTOS	Real-Time Operating System
S2MM	Stream to Memory-Mapped
SMC	Secure Monitor Call
SoC	System-on-Chip
SRAM	Static Random-Access Memory
SWaP-C	Size, Weight, Power and Cost
TCB	Trusted Computing Base
TMR	Triple Modular Redundancy
TZASC	TrustZone Address Space Controller
TZMA	TrustZone Memory Adapter
TZPC	TrustZone Protection Controller
VE	Virtualization Extensions
VHDL	Very High Speed Integrated Circuit Hardware Description Language
VM	Virtual Machine
VMM	Virtual Machine Monitor

1. Introduction

In this technological era where the interconnectivity between everyday objects is rising, embedded systems are becoming more and more predominant in our lives, bringing us closer to the Internet of Things (IoT). The connection to the Internet that many contemporary devices are capable of establishing, enables the user to access remote sensor data anywhere in the world. Many of these devices, aside from their core functionality, have some non-critical features that require management, and the growing necessity for these heterogeneous environments around embedded system solutions has led to a rise in the complexity of these systems. To manage such increase in complexity while still being a Size, Weight, Power and Cost (SWaP-C) optimized system, virtualization has become a normalized practice. To further complement this approach, and given the great emphasis to flexibility and performance on some safety-critical environments, there is a growing investment in exploring Field-Programmable Gate Arrays (FPGA) assisted approaches[GPG+15, PRAM17].

Virtualization allows the coexistence of heterogeneous environments on the same hardware platform [RG05] while enforcing the crucial features mentioned above, and can be both software or hardware-based. Contrary to more classical approaches, where complex systems could have multiple embedded control units, virtualization allows modern single embedded devices to run critical background services concurrently with user-oriented applications.

TrustZone hardware architecture [ARM09] virtualizes a physical core as two virtual cores, providing two execution environments: the secure and the non-secure worlds. This feature enables a strong temporal and spatial isolation between the guest Operating Systems (OSes), since the active guests runs on the non-secure world, while the inactive guests remain in the secure world. ARM TrustZone technology can be used to implement a virtualization solution with low overhead and low footprint [POP+14], with hypervisors that introduce a virtualization layer to the system stack, which enable the management of different instances of OSes, arbitrating their execution and resource usage, according to the chosen policy.

Furthermore, the wide spread of ARM processors also helps on making the porting solution easier.

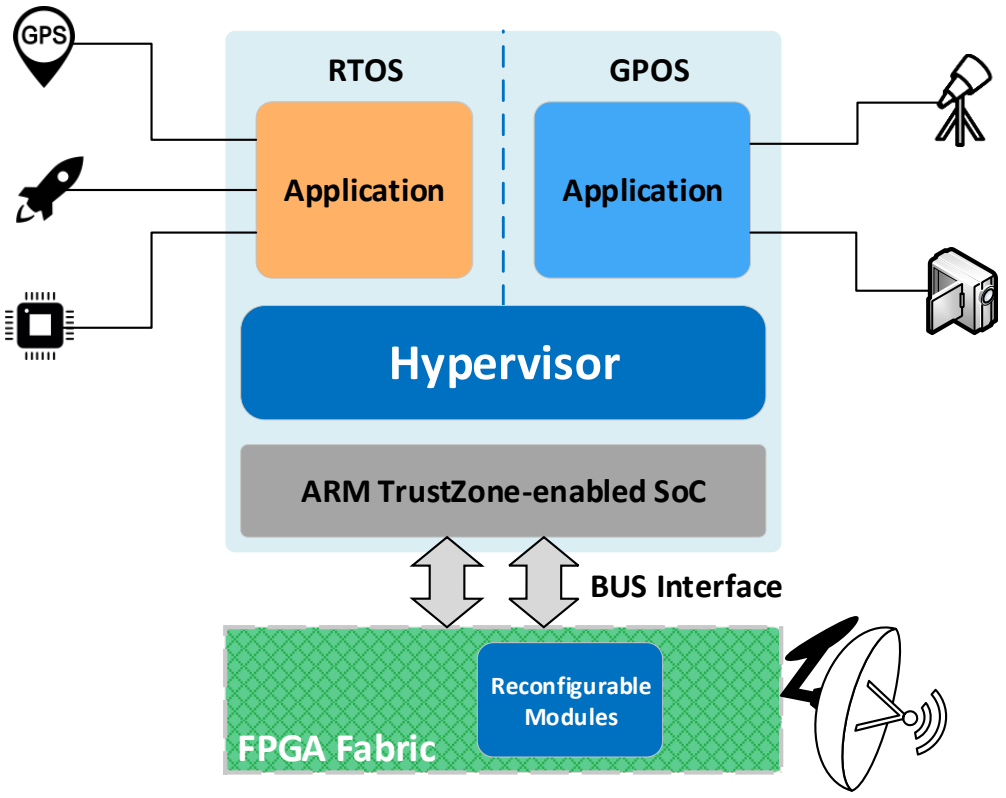


Figure 1.1: Generic TrustZone architecture for Space Applications.

Taking the example of a satellite operating system as seen in Figure 1.1, hypervisors can introduce several benefits to the system by allowing the incorporation of multiple OSes into a single hardware platform. The communication and control signals of the satellite are the critical tasks that cannot fail as they are essential to keep in contact with ground control and managing the satellites' telemetry and, as such, they should be run on a Real-Time Operating System (RTOS). Aside from that, some satellites have additional features like live video feed that is broadcasted on Internet platforms, which is a non-critical task and can be run in a General Purpose Operating System (GPOS). Having one hypervisor managing these OSes in a single platform reduces the amount of hardware necessary, which consequently reduces the risk of hardware fails.

Another technology that has been explored in recent decades, and is prone to use in space applications and hypervisors that have hardware modules, is Partial

Reconfiguration (PR) [Xil17] given the reconfigurability that it brings during runtime. This allows the system to have its functionalities changed during runtime using Dynamic Partial Reconfiguration (DPR), without needing to reconfigure all the hardware. In space, where high radiation can actively lead to the misbehaviours of hardware components, PR shines in critical hardware modules since, when a fault is detected, it allows the hardware module to be reconfigured. As with any system during useful life, components exhibit a failure rate, and to help solving this problem, DPR can ensure the system reliability for longer. Triple Modular Redundancy (TMR) is already a common practice to detect faults in critical, not tolerable to fail modules. If any of the three modules fails, the other two can correct and mask the fault. If a module is detected to consistently fail, it can be replaced using DPR, maintaining the system properly working. Another DPR benefit is that it allows to expand the security spectrum of the hypervisor hardware system by implementing security by diversity. This means implementing the same functionality using different implementations. When the system is detected to be under attack, the modules can be reconfigured through DPR changing their behaviour, while still obtaining the same results. Even if an attacker identifies the configuration of one implementation, changing that implementation during runtime will change the module footprint and force the attacker to discover the new behaviour of the hardware module.

Therefore, an hypervisor-based system that supports DPR, can bring improvements to the overall system,in performance, reliability and security.

1.1 Objectives

With the aforementioned points in mind, the proposal for this thesis is to create a mechanism to support the use of partial reconfiguration for μRTZVisor, which is a bare-metal TrustZone-assisted embedded hypervisor [MAC+17], exploring the potential benefits of a microkernel architecture and object-oriented implementation. The hypervisor will be extended with a software-hardware co-designed DPR framework focused on managing the reconfigurations and assignment of hardware accelerators according to requests of guest OSes.

A user-level software module will implement a DPR Manager, responsible for providing appropriate responses to the multiple requests for hardware accelerators from guest OSes, supported by a hardware controller module directly connected to each Reconfigurable Partition (RP) that will provide information about the current execution state and configuration of each RP.

This way, the FPGA resources can be treated as user applications, making the simultaneous management of hardware and software tasks possible in the hypervisor. Therefore, the DPR Manager will allow μRTZVisor to support reconfiguration of multiple RPs dynamically. The system will focus on the following goals:

- **Hardware Tasks**: As DPR allows to reprogram part of the FPGA while the rest of the hardware system is still running without interference, it is possible to introduce hardware tasks. Hardware tasks can be, for example, tasks which were formerly executed in software by the guests, implemented as hardware accelerators. A set of RPs will be used as containers for the hardware accelerators. In this system, hardware tasks are predefined bitstream files which hold the module fabric information for the desired hardware accelerator and have a target RP. The accelerators might be provided by the Manager itself or optionally by the guest that made the request. Different hardware tasks are dispatched by programming the assigned RPs with the respective bitstream file;

- **Low Reconfiguration Overheads**: As with most systems, one of the main requirements is to limit the overhead introduced by services as much as possible. To do this, the proposition is to implement the fastest reconfigurable method possible in the selected hardware platform, and couple it with fast responses to the guest OSes requests. A set of interrupts will also be implemented to send important signals to the guest, such as the end of execution by the module. This way, the guest OS is free to run other software tasks during the hardware reconfiguration and execution.

In summary, the proposed solution is intended to manage both software and hardware tasks, alongside the capability to reconfigure the modules when the need appears as fast as possible. The mechanism implemented and integrated into μRTZVisor hypervisor and was deployed in a Xilinx Zynq-7000 System-on-Chip (SoC) [Xil18b], for which the hypervisor was specifically tailored, will be under extensive evaluation.

1.2 Document Structure

The remainder of this thesis is organized as the following: Chapter 2 introduces the general techniques and concepts used throughout this thesis, first focusing on virtualization technology in embedded systems, dealing especially with real-time constraints. Then, the concepts and principles of DPR technology are described and the state of the art is presented, introducing their principles and major features. In Chapter 3, μRTZVisor is described in detail, which is the base platform where the proposed architecture will be integrated, while also describing the constraints given by the chosen hardware platform, the Zynq 7000. In Chapter 4, the proposed DPR mechanism architecture is explained, presenting the management of DPR modules in the system, describing both sharing and security mechanisms. Fundamental structures and design considerations are presented and explained. In Chapter 5, the evaluation of the proposed solution is presented, using standard open-source benchmarks as well as custom experiments are presented. The performance of the system in terms of real-time scheduling and reconfigurable computing are given and analysed. Conclusions and future perspectives are summarized in Chapter 6.

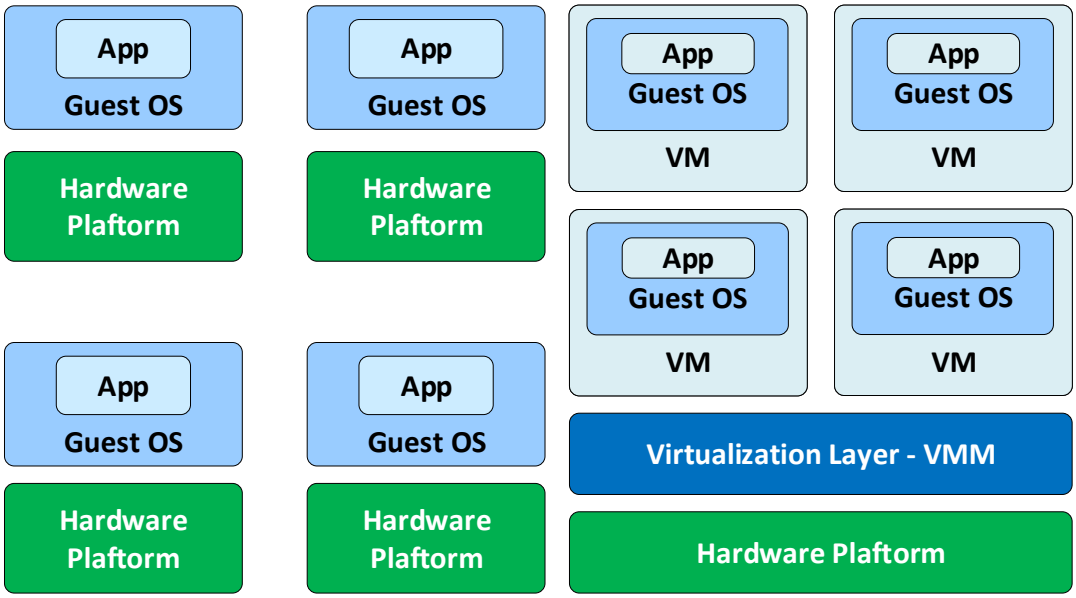
2. Literature Review and Related Work

This chapter aims at providing the necessary background to contextualize the work of this thesis. The concepts of virtualization, hypervisors and their different architectures, TrustZone and Partial Reconfiguration are introduced and described. Throughout this chapter, the state of the art in embedded virtualization is presented, focusing on partial reconfiguration systems and highlighting the developed solutions for the matter.

2.1 Virtualization

Traditionally, embedded systems were dedicated single-purpose systems characterised by hardware resource constraints and real-time requirements, but, with the growing computing functionalities and resources on general purpose platforms, systems that were formerly divided to provide different functions are now merging into one SoC [Hei11]. This is depicted in Figure 2.1, where 2.1a represents the more traditional approach with multiple machines running a single application and 2.1b represents a virtualized solution, with one machine running every application. Virtualization is a technology that allows the coexistence of multiple heterogeneous environments on the same hardware platform, providing an environment that abstracts the underlying hardware and enables the safe sharing of the available resources [Hei07]. The technology is well established in the enterprise and cloud computing space, but also presents huge benefits for today's intelligent portable devices such as smart-phones and vehicles, as it provides the advantages of better energy efficiency, shorter time-to-market cycles, higher reliability and overall service consolidation [AH10, SBM+16, Hei08].

Abstraction is achieved by introducing a software layer, called Virtual Machine Monitor (VMM) or hypervisor [Kai09]. This layer is what allows to have multiple Virtual Machines (VMs) [SN05] running in the same hardware with improved



(a) Tradional System. **(b)** Virtualized System.

Figure 2.1: System Evolution.

system safety and security, as the isolation and independence amongst different system components is assured, with any malfunction or incorrect behaviour being constrained to their domains and blocked from propagating to others. The virtual and physical computing platforms are decoupled via the VMs, which guarantee that each hosted guest has a separate and secure execution environment [Gol74].

The foundations for virtualization were introduced by Popek and Goldberg [PG74], when they identified the three essential properties for VMs:

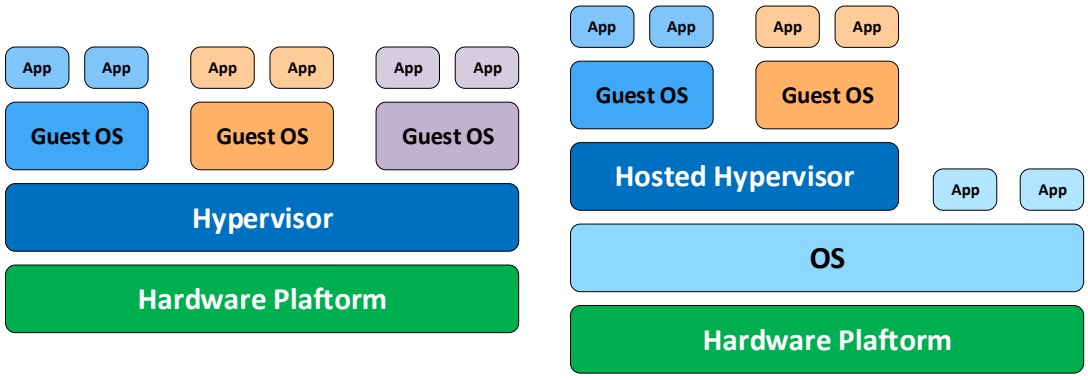
- Equivalence: the environment that each VM runs should be the as identical to the original or the emulated machine as possible, ideally allowing native OSes to be used directly in VMs without any modifications. If not possible, minimizing the costs of porting guest software to the VM is required.
- Efficiency: the performance from the incorporated OS must not be severely affected. For this to be achieved, the VMM must have the lowest overhead mechanisms as possible, while also improving the system's safety and security, providing environment isolation.
- Resource Control: the VMM must be in complete control of all system resources making it impossible for an arbitrary guest OS to alter resources from other guests. The machines should be thoroughly temporally and logically isolated.

Inevitably, while being hosted in a virtualized environment, the performance of guest software will be degraded, since the resources are shared with others, and as such meeting this three properties is required. They enable the porting of multiple guests to the solution with minimal engineering efforts while enforcing their isolation and keeping most of the native performance.

2.1.1 Hypervisor Types

A hypervisor or VMM, manages different instances guest OSes, running on the same platform, allowing them to seemingly run in an isolated environment by arbitrating their execution according to the chosen scheduling policy, and this brings performance and safety benefits to the system.

It is possible to distinguish two different types of hypervisors (Figure 2.2) based on the position of the virtualization layer in the system stack, or based on the permissions the VMM has to access hardware.



(a) Type 1 Hypervisor.

(b) Type 2 Hypervisor.

Figure 2.2: Types of Hypervisor Architectures.

Type-1 - bare-metal hypervisors (Figure 2.2a), have direct access to the hardware layer and manage the execution permissions of every system component, which means that all the hardware accesses are to be mediated and controlled by the VMM. As a consequence of this being the most privileged software on the running platform, the performance degradation of guests OSes will only be influenced by the performance of the hypervisor itself, making this type of hypervisor more suited to systems that must meet time constraints.

Type-2 - or hosted Hypervisors (Figure 2.2b), do not run directly above the hardware layer. Instead, they run on a OS that is already executing. This type of VMM usually does not have permissions to access and perform any operation on the hardware directly, since those responsibilities usually rest in the OS that runs

below the VMM, which usually results in lower performance ratings compared to type-1 hypervisors.

Regardless of the VMM type, the VMs must behave in the same way as they would if they were executing directly over the hardware platform. In the domain of embedded systems, virtualization solutions are mostly bare-metal hypervisors, because having an OS between the hypervisor and the hrdware will introduce an additional overhead to the system.

There are two different approaches, applicable to both hypervisor types, towards a virtualization solution: full virtualization and para-virtualization. The full virtualization technique is also called native virtualization [HRL+08, Kai09]. In this kind of virtualization, guests OSes require no software modification and rely on the VMM to emulate the low-level features of the hardware platform. This feature allows native OSes like Linux or Android to run directly inside the virtual machines. Since it does not rely on OS code, even closed-source software can be easily hosted. This technique relies on supporting technologies as virtualizable Instruction Set Architecture (ISA) and hardware extensions, to support unmodified guests and seemingly make them not aware they are not running on the actual hardware.

Para-virtualization [KK12, Kai09], on the other hand, refers to communication between the guest software and the VMM to implement virtualization, with guests being aware that they are virtualized to take advantage of the hypervisor features. This mechanism is mostly implemented by modifying the guest software code to interact with the hypervisor's Application Programming Interfaces (APIs). This type of virtualization is especially suitable for architectures without hardware assistance.

2.1.2 Kernel Architectures

The kernel is a software component in any OS that runs with the highest privileges, having no restrictions over the entire system, in addition to being part of its Trusted Computing Base (TCB). Ideally, kernels should be as small as possible to minimize the frequency of bugs and reduce the TCB's attack surface, in order to become fully trustworthy [Kai09, HPHS04, HEK+07].

The classical approach to this problem are monolithic architectures, where all the modules are linked together in a common address space and execute in the most privileged mode [KK12, THB06]. Figure 2.3 is an example of a monolithic implementation of a hypervisor, where all the OS functionalities like interrupt

handling, memory management, device drivers, network stacks, Inter-Partition Communication (IPC) facilities and file systems are implemented inside the kernel.

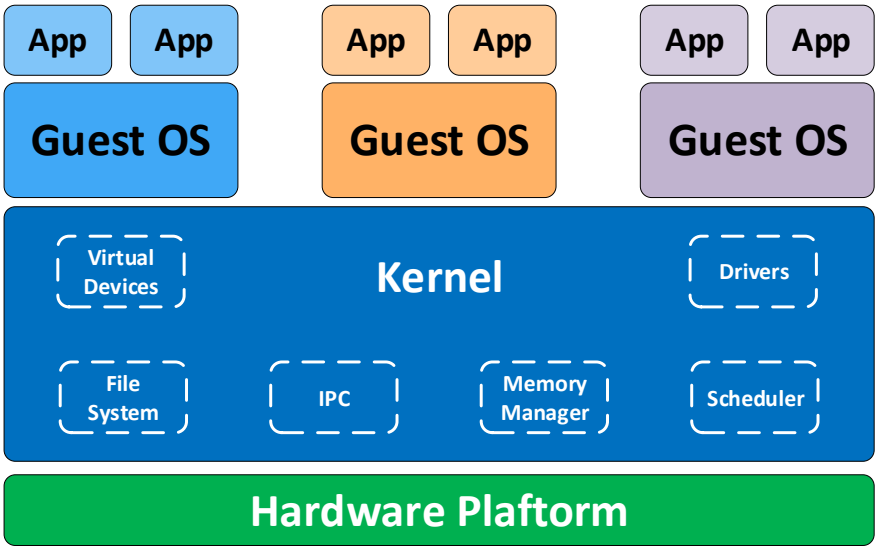


Figure 2.3: Monolithic Architecture.

The major advantage of monolithic architectures is that, as all the services reside in the same address space, the interaction between subsystems is fast and efficient. The advantage of good performance is coupled with the disadvantage of a larger TCB, since all code runs with complete access to all system resources and must, therefore, be trustworthy. In widely used OSes, such as Windows or Linux, which are colossal systems in terms of lines of code, the number of bugs and vulnerabilities increases vastly.

An alternative approach to monolithic kernels are microkernel architecture where the objective is making the kernel as small as possible. Device drivers and other services are migrated to user-level servers, in order to minimize the TCB and thus the attack surface [EH13]. As seen in Figure 2.4, only the memory management, scheduling module and IPC are kept under kernel space, which can be seen as a minimalistic implementation.

As the main drive with microkernel architectures is to reduce the amount of privileged code, there are no real differences between a system server and an application, since all run in user mode. User applications access system servers through IPC mechanisms, which become performance-critical in the architecture [HHL+97]. This induces a handicap to the microkernel approach because of the obligatory reliance on IPC infrastructure, which was the main reason for microkernels to not be accepted as a valid solution for several years [Lie96].

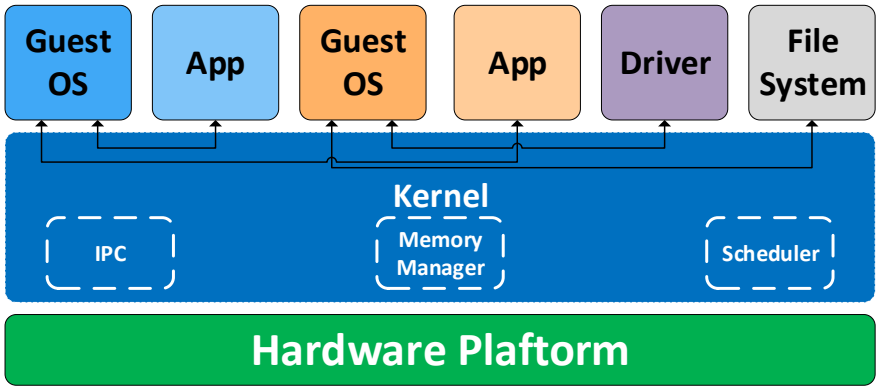


Figure 2.4: Microkernel Architecture.

Microkernel architectures are proven to be more secure than a monolithic kernel, given the benefit that if a service server fails, the hypervisor kernel remains unaffected, which in a monolithic system could cause the system to be blocked.

2.2 ARM TrustZone

ARM TrustZone technology [ARM09] refers to the security extensions that were introduced with the ARMv6 architecture. The ARM1176JZ(F)-S processor was the first of the TrustZone processors, with the hardware architecture aiming at providing a security framework that enables the device to counter many of the specific threats that it will experience. TrustZone technology enables system-wide security by integrating protective measures into the ARM processor, bus fabric, and system peripheral Intellectual Property (IP).

The TrustZone hardware architecture virtualizes a physical core as two virtual cores, providing two execution environments: the secure and the non-secure worlds (Figure 2.5). The major change introduced in the hardware architecture is the ability to tag system resources as belonging to the secure or normal world. To indicate in which world the processor is executing, there is the new 33rd processor bit - NS (Non-Secure) bit, which is also extended to the rest of devices, enhancing control for the system designer over peripheral buses and memory [Win08], [SRSW14]. To preserve the processor state during the world switch, TrustZone adds an extra processor mode: the monitor mode, which always executes in the secure state, independently of the value of the NS bit. Since the processor only runs in one world at a time, software stacks in both worlds can be bridged via a new privileged instruction - Secure Monitor Calls (SMCs). The monitor mode

can also be entered by configuring it to handle interrupts and exceptions in the secure side. To ensure a strong isolation between secure and non-secure states, some special registers are banked, while others are totally unavailable to the non-secure side. The memory infrastructure outside the core can be partitioned into the two worlds through the TrustZone Address Space Controller (TZASC) and the TrustZone Memory Adapter (TZMA), allowing distinct memory regions to be configured and used in one or both worlds. TZASC can partition the Dynamic Random-Access Memory (DRAM) into different secure and non-secure memory regions, by using a programming interface which is only accessible from the secure side, while the TZMA provides similar functionality but for on-SoC Read-Only Memory (ROM) or Static Random-Access Memory (SRAM).

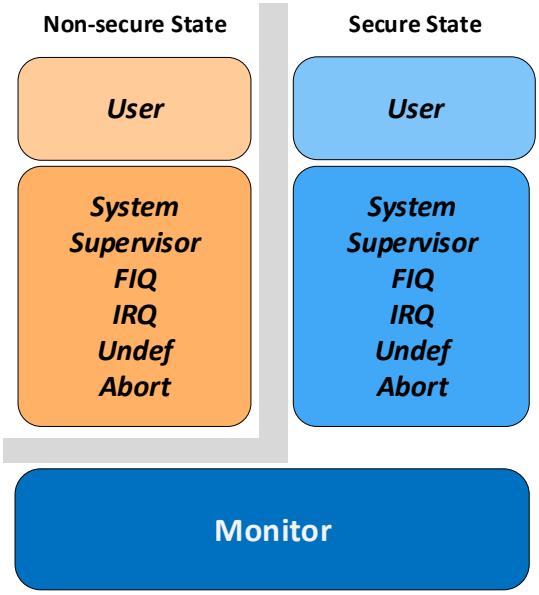


Figure 2.5: Modes of an ARM core implementing TrustZone.

System peripherals can be also configured as secure or non-secure through the TrustZone Protection Controller (TZPC).

2.2.1 TrustZone-based Virtualization

Despite the fact that TrustZone extensions are not oriented towards virtualization solutions, being a security-oriented technology, the ability to have full control over the exception system and the different execution levels that can be used for both guest OSes and hypervisor, means that it provides similar features to those offered by Virtualization Extensions (VE), and is viable for use in virtualization

[PS18]. However, TrustZone does not fulfil the requirements of classical virtualization, as for example, one of the requirements that needs some addressing is memory management. Although, guests can run nearly unmodified, they need to be specially compiled to execute in the confinement of their attributed segments, once TrustZone does not provide two-level address translation, only providing memory segmentation support through peripherals such as TZASC.

Despite the drawbacks, the extension has been explored in embedded virtualization, given the ubiquity of ARM platforms in everyday devices potentiating the easy use of the technology. Typically, existing TrustZone-based hypervisor solutions will execute the hypervisor in monitor mode while guests run in non-secure state to have full leverage of all the facilities. The most common implementations are usually monolithic hypervisors, supporting a dual-OS configuration, due to the perfect match between the number of guests and virtual states supported by the processors, as shown in Figure 2.6.

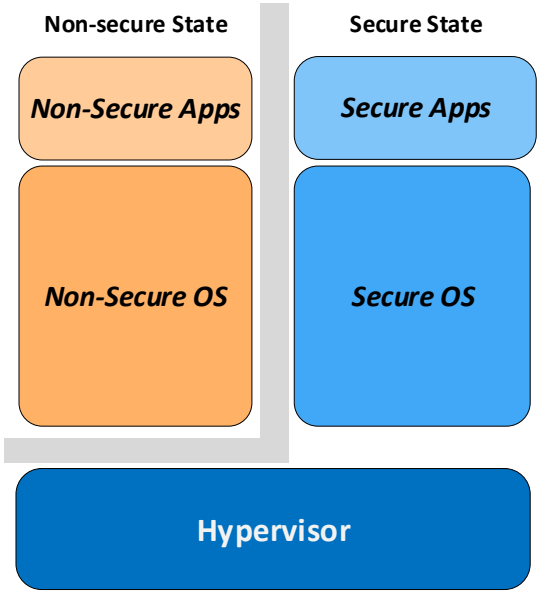


Figure 2.6: Generic ARM TrustZone architecture adaptation suitable for Hypervisors.

Over the last few decades, several works have been carried out towards finding viable frameworks for embedded systems using TrustZone technology. In [Win08], Winter introduced a virtualization framework for handling non-secure world guests, which was based on a secure version of the Linux kernel that was able to boot only an adapted Linux kernel as non-secure world guest. Later in [FLWH10], Frenzel proposes the use of TrustZone technology to implement the

Nizza secure architecture, which consists of a minimal adapted version of Linux kernel (as normal world OS) on top of a hypervisor running on the secure world side. Those first solutions were followed by SafeG [SHT13], which implements a dual-OS configuration capitalizing on TrustZone worlds, where an RTOS for time-critical tasks is running on the secure side while a GPOS runs on the non-secure world. In a similar implementation, the Secure Automotive Software Platform or SASP [KLJ+13] is a virtualization framework based on TrustZone that allows the infotainment system of a vehicle to be simultaneously managed alongside the critical control system, guaranteeing secure device access. VOSYSmonitor [LCP+17], also enables concurrent execution of two OSes, a safety critical RTOS and a GPOS. However, this implementation distinguishes itself from others, because it is implemented over ARMv8-A processors. In the in-house developed family of hypervisors, the TZVisor family, LTZVisor [PPG+17], which is an open-source lightweight TrustZone-assisted hypervisor mainly targeting the consolidation of mixed-criticality systems, also supports the coexistence of two OSes: a secure RTOS running concurrently with an untrusted GPOS.

Nonetheless, some TrustZone hypervisors support more than two guests, namely RTZVisor [PTM16] and μRTZVisor [MAC+17], also from the TZVisor family, which are capable of running multiple guest OSes at a time by multiplexing them on the non-secure world.

2.3 Partial Reconfiguration

Partial Reconfiguration has been used to increase the flexibility of systems, on both standalone FPGAs and heterogeneous platforms that integrate Central Processing Units (CPU)s and FPGAs on the same platform. Since the focal point of the dissertation are heterogeneous platforms, they will be the main focus from now onwards, and in particular the platforms from Xilinx that integrate a Processing System (PS) and a Programmable Logic (PL) subsystems.

The purpose of hardware-based acceleration is to increase the system throughput by delegating modules outside the PS to perform former software tasks, providing more free time on the CPU side, so more tasks can be performed. The trade-off here results in more hardware resource usage and increased complexity in system design. It is relevant to say that not all of the overhead is taken off the PS, since communications procedures between the hardware accelerator and the PS are still needed and may vary in time with procedures such as handshakes, data transaction and validation of signals. In order to justify the hardware modules,

most of the times, an accelerated task must be faster than its equivalent software task.

To solve specific problems, a custom hardware accelerator is more efficient than a general-purpose computer, and custom-designed accelerators can be implemented into PL. Designing an application-specific hardware accelerator can significantly increase the computational efficiency by exploiting the parallelism in an algorithm, as well as removing resource intensive tasks from the CPU processing time [LOG+03]

FPGAs are reconfigurable hardware chips that can be reprogrammed to implement varied combinational and sequential logic blocks and can still be considered low cost compared to Application Specific Integrated Circuits (ASICs) with the advantage of being quickly reusable. Their reprogramability offers great flexibility and the opportunity to quickly develop a prototype of a circuit. While not as fast as ASICs, FPGAs have an advantage in low volume prototyping and proof-of-concept applications. FPGAs allow hardware designs to be quickly and cheaply validated and offer a scalable solution for performance-limited systems [HN06].

However, while still providing the parallelism ability, traditional FPGA reconfiguration had one major drawback - the lack of flexibility in the reconfiguration, once the whole fabric was required to be reconfigured even when the smallest modification was necessary. Today, some FPGA vendors, like Xilinx and Altera, enable run-time reconfiguration in their architectures. RP allows the system to be designed to have special reconfigurable regions in the FPGA that can be dynamically reconfigured while the remaining of the FPGA design continues to execute normally, as exemplified in Figure 2.7. As such, PR has been trending in the scientific community for the past decades [BHH+07], becoming a interesting topic towards cloud computing [BSB+14] and space applications [McD08].

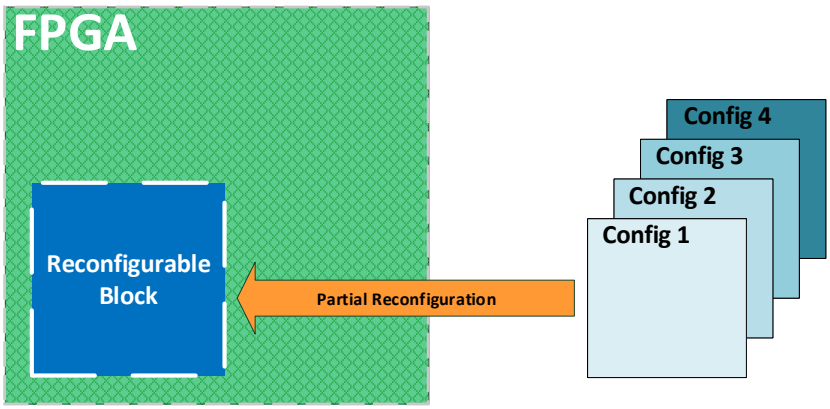


Figure 2.7: Partial Reconfiguration Overview.

As PR enables the dynamic utilization and allocation of resources, designers can implement more complex algorithms breaking them down into smaller modules and reconfigure them as they complete their tasks. This eliminates the need for larger logic areas. It also allows the system's power consumption to be reduced when compared to full reconfigurations each time a functionality change is required [TCL09].

For each configuration implemented for a reconfigurable block, a partial bitstream will be generated and his used during the runtime to reconfigure the block. Partial bitstreams are smaller than full bitstreams and their reconfiguration time is shorter, which leads to less processing time and energy consumption during the reconfiguration. Other feature enabled by the technology, is that it can be used to reconfigure the regions with blank bitstreams while they are inactive and not necessary, as long as the throughput for the configuration is high enough [LPFG10], reducing power consumption.

Several researchers in the last decade have been searching for a truly efficient dynamic reconfiguration framework [HGNB10], as well as power optimization [LKLJ09, BBCS14]. However, the technology is not yet suitable for most real-world applications as it suffers from a somewhat still expensive reconfiguration overhead, which is still a drawback [McD08]. In a computing-intensive system, where several mutually exclusive components are sharing reconfigurable resources, the time lost on reconfiguration will severely degrade the overall performance [HD07] even with carefully executed designs.

2.4 Related Work

In this section, state of the art implementations of DPR frameworks, focusing on projects making use of runtime reconfigurable hardware platforms in order to maximize the performance of the system. Related works which target OSes and hypervisors that employ similar features to ours, and implementations with the goal of maximizing the reconfiguration throughput for run-time reconfigurations, are detailed.

One of the concepts that promoted the raise in popularity of frameworks supporting PR, was OSes having core modules accelerated in hardware. OSes providing their software features implemented as hardware accelerators capable of concurrent processing with the software, some going further by having a complete hardware task system, provide great benefits in performance while considerably reducing the workload on the PS.

2.4.1 ReconOS

The key idea behind ReconOS [LP09] is providing the structural foundation to support a multi-threading programming model across the hardware and software in SoCs armed with FPGAs. The implemented model, aside from the multi-threading, provides interfaces, communication channels, such as message queues and semaphores, memory access and address translation to the hardware threads, allowing fast integration of new hardware accelerators in the system. Also, the system comprises software components in the form of libraries and kernel modules that offer an interface to the hardware, the operating system, and the application's software threads.

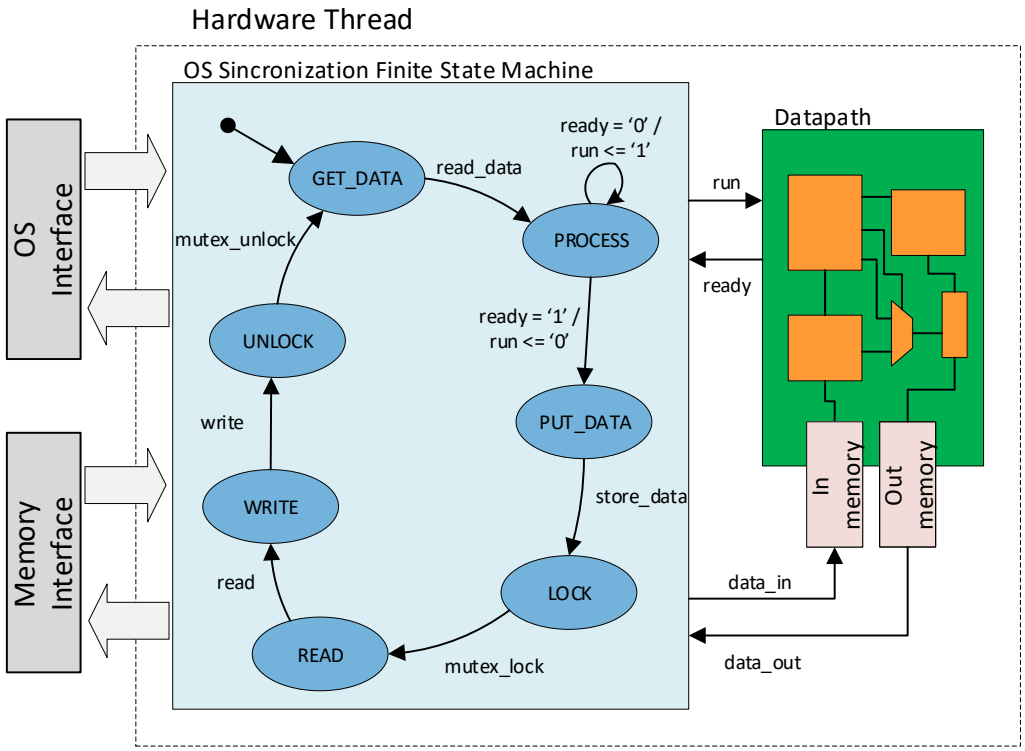


Figure 2.8: ReconOS Hardware Threads synchronization with OS. Adapted from [LP09].

Hardware accelerators are placed in reconfigurable slots as hardware threads, which are predefined areas of reconfigurable logic equipped with the necessary communication interface, while software threads alongside with the RTOS kernel are executed on the devices' embedded PS. On ReconOS, every hardware thread is independent from the software tasks, and has access to every service of the OS which allows for parallel execution of hardware and software threads. To enable hardware threads to access services, such as semaphores or mutexes, the

corresponding software API calls are exposed to the hardware through a Very High Speed Integrated Circuit Hardware Description Language (VHDL) library.

The management of every hardware thread is done via a custom designed OS Synchronization Finite State Machine (OSFSM) which is connected to the hardware logic of the accelerator and both the memory, through the Memory Interface (MEMIF) and the OS, via the Operating System Interface Module (OSIF). The aforementioned VHDL library is used by the OSFSM to manipulate the interfaces.

Figure 2.8 depicts how the synchronism between the hardware and the OS is established. At the beginning of an execution cycle, the hardware thread awaits for the read_data semaphore. When the semaphore is given, the thread reads the data from the local DRAM and consequently processes it. It then signals the OS that the data is ready to be stored, operation which is dependent on the MEMIF. When the cycle is complete, the thread becomes available again and returns to the await semaphore stage.

When implemented in Xilinx FPGAs, DPR via the Internal Configuration Access Port (ICAP) port is used to reconfigure the hardware accelerators during run-time. Using the ICAP port instead of the more traditional Processor Configuration Access Port (PCAP) allows the hardware to be reconfigured while the software keeps executing, instead of stalling the operating systems during the bitstream transfer, while also lowering the times required for each configuration. However, the PCAP is the most commonly used as it does not require any hardware resources but is limited to the theoretical 145MB/s of throughput, which is the slowest reconfiguration interface on the platform. The differences between the PCAP and ICAP will be detailed in section 3.2.

2.4.2 FRED

Another similar project to this work is the FRED framework [BBP+16], which consists of a heterogeneous computing system composed by a PS and a DPR-enabled FPGA fabric, both sharing a common memory, supporting both software and hardware tasks. Hardware tasks are, similarly to the other projects, hardware accelerators that can be configured and executed on the FPGA.

The FRED framework proposes that software tasks can have parts of their computation accelerated by requesting the execution of hardware tasks to an entity responsible for managing the hardware. This is where the proposed framework enters ensuring predictability when reconfiguring the FPGA while minimizing the overhead related to hardware allocation. Each software task may only be connected to one hardware task at a given time, despite the FPGA being partitioned in

multiple regions containing multiple hardware tasks. In the same line of thought, one hardware task can only be used by a single software task and can execute only if it has been programmed into one of its slots. Hardware tasks execute in a non-preemptive manner being that once started the execution related to one hardware task, they will execute until they finish the process.

The implementation uses an FPGA Reconfiguration Interface (FRI), which is a peripheral device external to the processor in a similar way to a DMA and hence does not consume processor cycles to reconfigure slots in runtime. As an example, Figure 2.9 shows one software task using two hardware tasks consecutively while being descheduled after requesting the use of the hardware task until the execution is complete.

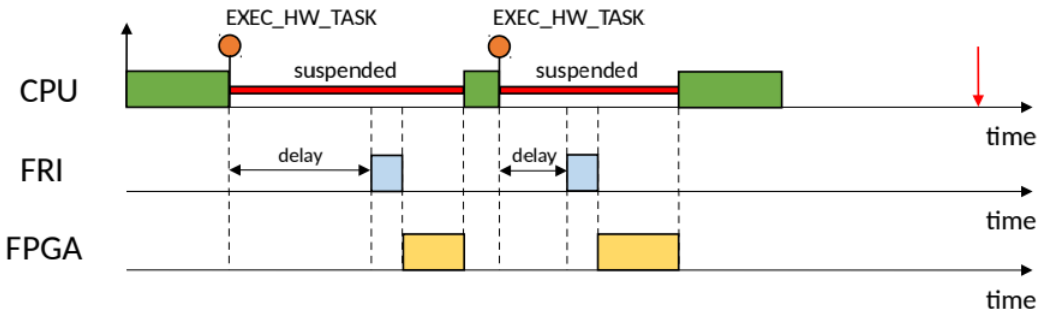


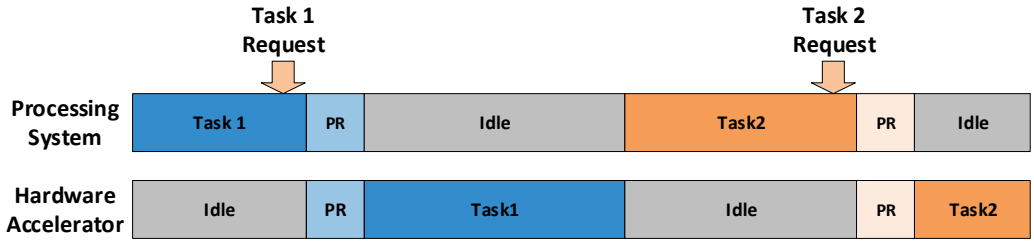
Figure 2.9: Sample schedule of a Software task using two Hardware tasks [BBP+16].

The scheduling mechanism is implemented in a multi-level structure of queues, composed by software partition queues and the FRI queues. The partition queues are ordered according to the First-In-First-Out (FIFO) policy. Each time a software task issues an execution request for a hardware task, is assigned a ticket marked with the current absolute time. Then, is inserted into its corresponding partition queue (depending on the affinity of the hardware task). The partition queues enqueue a request as long as there are no free slots into the corresponding partition. The FRI queue is fed by the partition queues and is ordered by increasing ticket time. This mechanism guarantees predictable delays incurred by hardware task requests,

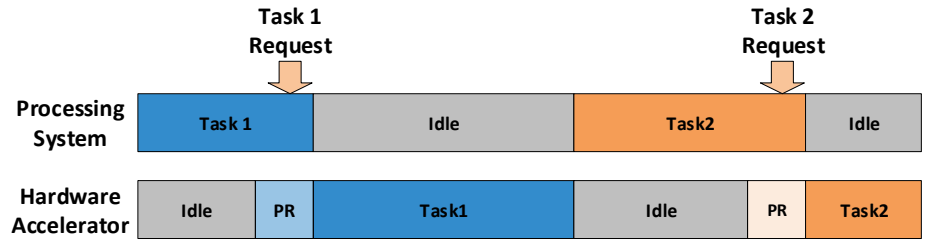
The communication between the tasks is done by shared-memory. In contrast to other approaches that store the input/output data into private memory areas within the FPGA slots, the solution adopted in FRED allows decoupling the time a hardware task must hold a slot from the scheduling delays of software tasks. This property facilitates the derivation of bounds on the delay incurred by the software tasks when requesting hardware tasks.

2.4.3 ZyCAP

The ZyCAP [VS14] is a custom PR controller designed to achieve maximum performance on hardware runtime reconfigurations in Xilinx platforms and to accelerate the integration of PR in custom made hardware designs, using the ICAP port, which allows the fastest reconfiguration times on these platforms. ZyCAP is an open-source hardware IP that is composed by a soft Direct Memory Access Controller (DMAC), a custom made ICAP manager and the respective connection to the ICAP and an associated driver, which was implemented having in consideration the minimal workload possible for the PS. The hardware module uses two interfaces with the PS, an AXI-Lite interface to exchange the control signals and an Advanced eXtensible Interface 4 (AXI4) interface connected to an AXI HP port where the partial bitstreams are fetched to hardware.



(a) PCAP Reconfiguration Flow.



(b) ZyCAP ICAP based Reconfiguration Flow.

Figure 2.10: Reconfiguration Approaches. Adapted from [VS14].

The maximum performance was achieved using the Direct Memory Access (DMA) for the transfers of the bitstreams to hardware. The controller is configured with the starting address and size of the partial bitstream through the AXI-Lite interface and bitstreams are transferred from the DRAM to the ZyCAP controller at high-speed through the AXI HP, port using the burst transfers enabled by the AXI4 interface. The custom ICAP manager converts the streaming data received from the DMA controller to the required format for the ICAP port. Finally, an

interrupt is used to notify the software when the transfer and the reconfiguration are complete.

ZyCAP achieved a maximum reconfiguration throughput of 382 MB/s (95.5% of the theoretical maximum using the ICAP interface accelerated with the AXI DMA, improving over the Xilinx provided IPs and PCAP significantly. The deviation from theoretical maximum is due to the software driver overhead, mainly when doing the DMA configuration for the bitstream transfer and interrupt handling, but also is caused by the DRAM access latency.

Figure 2.10 shows the differences and the gains obtained using the ICAP based approach. The ICAP based controller (Figure 2.10a), allows for the reconfiguration to be performed in parallel to the executing task without stopping the software execution, as opposed to the PCAP based controller (Figure 2.10b) which needs the CPU to coordinate the reconfiguration.

However, the ZyCAP was designed for a single RP and follows a predefined reconfiguration sequence with zero decision time associated. The project was implemented as a proof of concept and to be used in a multiple reconfigurable blocks system it would require several modifications.

2.4.4 CODEZERO DPR Platform

Another relevant implementation using DPR was made by Jain et Al. [PJC+13], where the researchers started with the CODEZERO microkernel hypervisor and modified it to virtualize and manage both hardware and software components of the platform. The hypervisor creates an abstraction of the underlying hardware platform so that it can be used by multiple guest OSes, managing and executing software and hardware tasks in the same manner, which makes hardware tasks viewed as another software task. The hypervisor was modified to execute on the dual-core ARM of the Zynq-7000 hybrid platform while providing support for hardware task execution and scheduling.

However, the classical DPR technology is not present in this implementation for hardware reconfiguration. Instead, reconfigurable computing components in this framework are implemented by intermediate fabrics regions, which are built of an interconnected network of coarse-grained processing elements overlaid on top of the original FPGA fabric.

Two scheduling mechanisms were implemented in the hypervisor for the hardware tasks - nonpreemptive hardware context switching and preemptive hardware context switching. In non-preemptive hardware context switching the scheduling

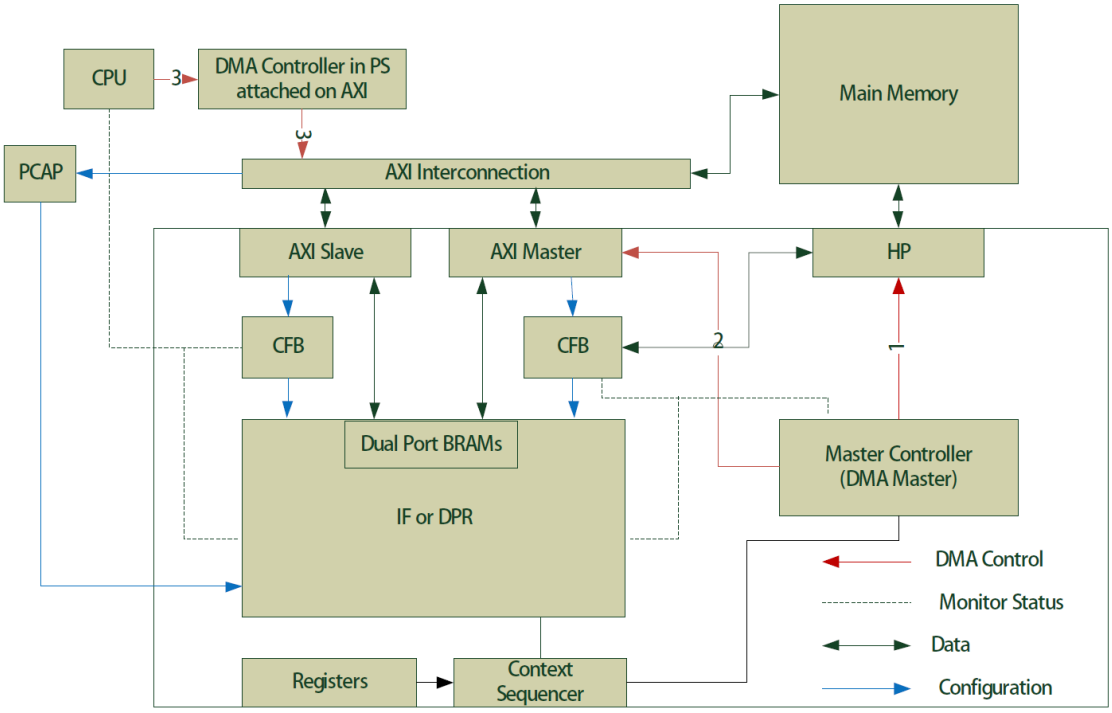


Figure 2.11: Block Diagram of the Reconfigurable Region [PJC+13].

only occurs when a hardware context completes. At the start of a context an interrupt that signals the start of execution is triggered and an hypervisor mutex, the L4 mutex control, locks the reconfigurable fabric. Another interrupt is triggered at the end of the execution so that the mutex is released and another context can be loaded. The advantage of non-preemptive hardware context switching is that context saving and restoring is not necessary, as the task scheduling occurs only after a context finishes, implying that minimal modifications are required in the hypervisor to add support for these tasks, as the existing hypervisor scheduling policy and kernel scheme are enough.

In the alternative pre-emptive hardware context switching, it is necessary to save a context frame and to restore it when necessary. Context-saving refers to a readback mechanism to record the current context, the status, the DMA controller status and the internal state. This information is stored in the task control block which is a similar way to saving the CPU register set in a context switch. A context frame restore occurs when a hardware task is swapped out, and an existing task resumes its operation. This approach would provide a faster response, compared to non-preemptive context switching, but the overhead is considerably higher. This requires modification of the CODEZERO's task control block data structure and the hypervisor's context switch mechanism, as well as a number of additional APIs.

The exchange of data between the PS and PL is done using the DMA method. The DMAC, being independent from the CPU, allows for transfers to be executed without creating any overhead to the PS. The DMA is triggered by the hardware, to store the data in the memory, when the hardware data buffer is full. Additionally, the PS can also request to read the data from the PL, programming the DMAC to do the read process.

The researchers tested the system where they demonstrated the functionality of the implementation with multiple different tasks running concurrently with the hypervisor providing the necessary isolation [JPC+14].

2.4.5 Ker-ONE

The Ker-ONE [Xia16] represents a re-design of the Mini-NOVA framework [XPN15a, XPN15b], while following the same design principles, into a new microkernel for the ARMv7 architecture. It was designed having in mind the coexistence of both software and hardware tasks, and supporting the possibility to reconfigure the accelerators in runtime using DPR. This allows that, for each Partial Reconfigurable Regions (PRR) placed on the FPGA, multiple unique algorithms can be implemented and DPR makes the swapping between functions possible. Aside from having multiple PRR running different functionalities, it is also possible to have the same algorithm implemented in more than one PRR. This means, when requested from a guest OS, the module can be implemented in any of the PRRs, making unnecessary for the system to have wait for a specific accelerator to be released.

Implemented on the Zynq-7000 platform, the reconfigurable accelerator modules are hosted and run in separate PRR, and are reconfigurable by downloading the configuration data through the PCAP. All the configurations necessary for each hardware task are sent from the PS to the PL through the AXI GPs. For the data exchange, the system uses the AXI HP ports. In order to protect the bitstreams from attacks, all of them are mapped in the memory space of the task manager. This module is responsible for assigning the most appropriate PRR when a specific hardware task is required by a guest OS. When a request happens it checks each PRR and selects the region to be used as well as reconfiguring it if necessary. Then the caller guest is allowed to use this hardware task as its client.

Figure 2.13 represents the states that each PRR might be at any time. As depicted, a PRR can only be directly allocated to VMs when it is in idle state and requires no reconfiguration. In other situations, the allocation process requires extra overheads caused by PCAP transfer or preemption.

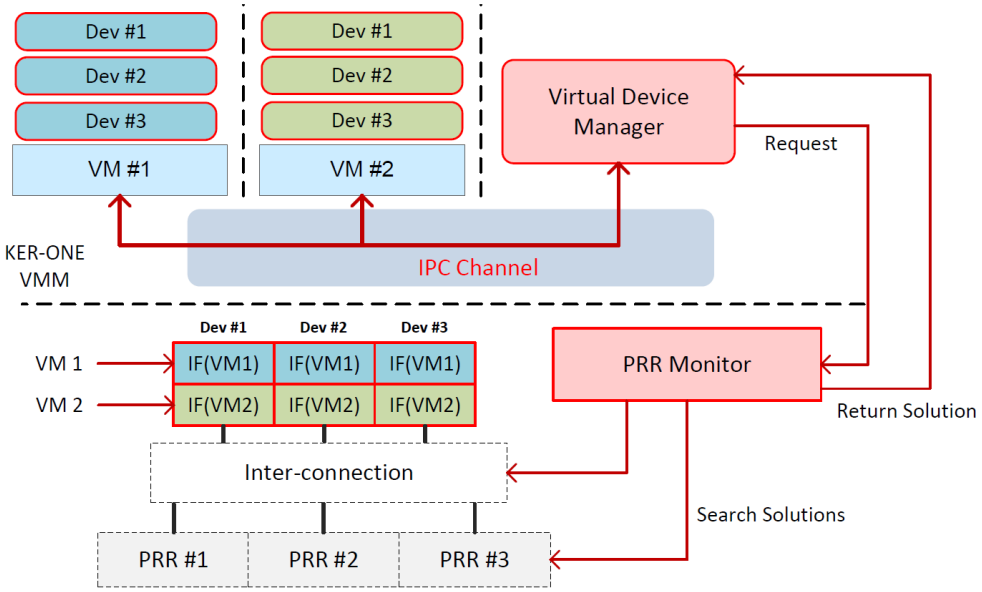


Figure 2.12: Overview of the DPR management framework in Ker-ONE [Xia16].

An extra security mechanism, hardware Memory Management Unit (hwMMU) was introduced, which guarantees that one hardware task is exclusively used once it is given to a specific guest OS and the hardware task should only access the data section of the VM which is currently using it, and accessing a memory space outside the specific section is forbidden. The hwMMU is implemented in the reconfigurable regions controller and is in charge of monitoring any access to the PS side. When a hardware task is allocated to one guest OS, the hwMMU is configured with the physical address of the guest OS's hardware task data section.

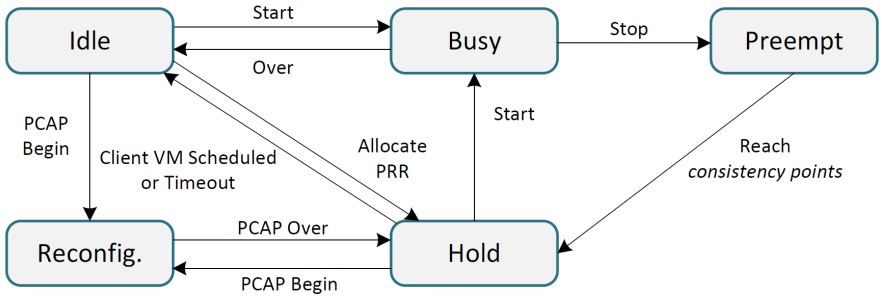


Figure 2.13: Ker-ONE Reconfigurable Regions State Machine [Xia16].

To know which VM is accessing one specific hardware task, the page tables where the configuration is kept are updated with information about the VM that is using it. The hardware task's behaviour is controlled by a PRR register group, which contains information about the current configuration and is mapped to its current VM client as the hardware task interface.

3. Platform and Tools

This chapter aims at providing the contextualization of the hypervisor and development board used for the development of the project. The μRTZVisor hypervisor and its architecture, as well as the Zynq 7000 board are introduced and described.

3.1 μRTZVisor

Designed to achieve better and improved levels of safety and security, the μRTZVisor is based on a refactoring of the RTZVisor [PTM16], which is a bare-metal embedded hypervisor that relies on TrustZone hardware to provide the foundation to implement strong spatial and temporal isolation between multiple guest OSes, from C to C++ exploring an object-oriented language promoting a higher structural degree and modularity to the implementation [MAC+17].

The μRTZVisor hypervisor, based on the microkernel architecture, differs from traditional implementations as instead of using para-virtualization techniques, which enforce guest source-code modifications, it was designed to keep the RTZVisor's capability of running nearly unmodified guest OSes while potentiating the benefits of improved security and design flexibility inherited from the microkernel implementation.

As TrustZone-enabled processors provide a secure and non-secure view of the processor, which is also extended to devices and interrupts, guest OSes are allowed to configure and managed their assigned system resources and interrupts. The main disadvantage introduced by the technology is that this OSes need to be compiled to be executed in the assigned memory segments. Another limitation is that the TZASC only allows for fixed memory sections to be configured as secure or non-secure. On the Zynq-7000, for the Random Access Memory (RAM) the granularity is 64MB, which limits the number of possible supported concurrent guests running on the platform.

Besides relocation, guest OSes only need to be modified if they are required to use auxiliary services or shared resources that rely on the kernel's IPC facilities. The overview of the hypervisor architecture can be seen in Figure 3.1.

Multi-guest support is achieved by dynamically changing the security state of the memory segments, devices and interrupts at each context switch. The active partition has its system resources configured as non-secure during runtime, while all the inactive partitions have theirs configured as secure. If an active guest tries to access secure resources, the hypervisor will receive an abort exception to be handled.

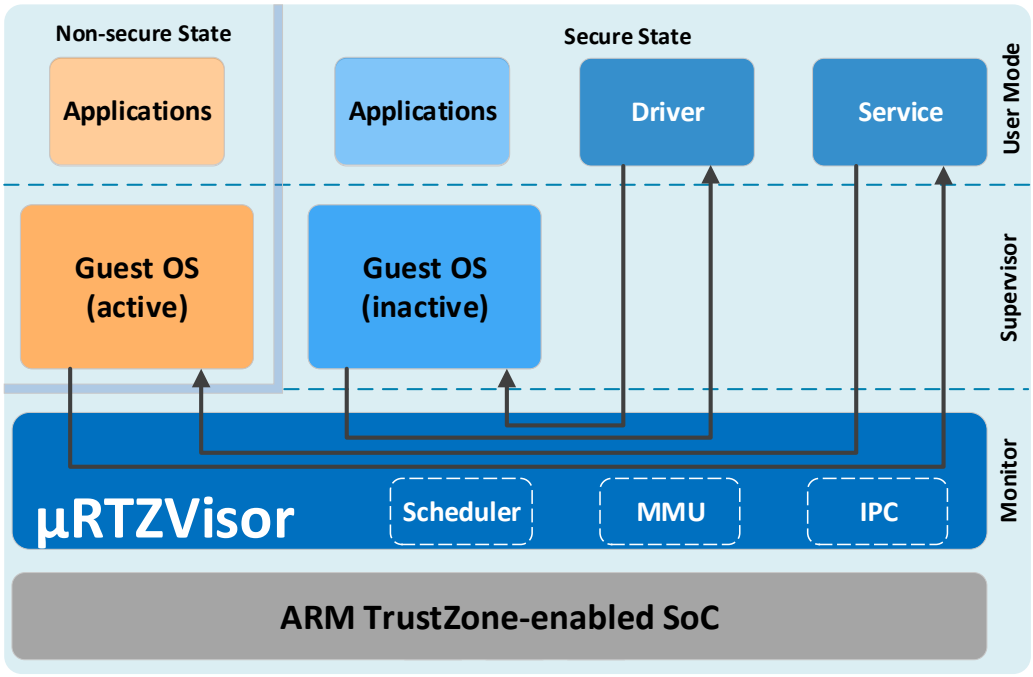


Figure 3.1: µRTZVisor architecture.

The hypervisor privileged code runs in monitor mode, which is the highest level of privilege in TrustZone, enabling the complete access and configuration rights over all system resources at all time. Following the principles of a microkernel architecture, only the most essential infrastructures should be implemented at this level, which is why only the IPC and the required mechanisms to implement the VM abstraction, spatial and temporal partitioning are implemented at this level. Secure user mode partitions are used to implement extra functionalities that usually are in the kernel of a monolithic system. In µRTZVisor, these functionalities are implemented as server tasks that can be accessed through Remote Procedure Calls (RPC), sitting on the IPC and scheduling infrastructure. They were implemented as secure world tasks in the hypervisor, since secure virtual address space

eliminates the need for relocation and recompilation while also reducing the fragmentation inherent to the segmented memory model. This also makes the process of adding, removing or replacing services easier, while also enabling finer-grained fault-encapsulation and making the propagation of faults to other components easier to contain.

A crucial design characteristic of μRTZVisor is that partitions are allocated statically, at compile-time which simplifies the implementation of partition management, communication channels and resource distribution. Aside from encapsulating the services, TrustZone's multiple privilege levels and control over memory segments, devices and interrupts provides is key in order to achieve a higher degree of security. Other hardware infrastructures such as virtual address translation and the capability based system that controls the access to diverse mechanisms present in the hypervisor also provide more robustness to the solution.

Each capability represents a kernel object or a hardware resource and a set of rights over it. A partition that owns a certain capability can access it according to the set read/write permissions. This means, that even secure tasks have to configure and interact with their assigned hardware resources in an indirect manner, by issuing hypercalls to the kernel, and makes the use of a system resource by partitions conceptually impossible if they do not own a capability for it. The capability system overall provides greater control over resource distribution besides reducing system configuration almost only to the this flexible mechanism.

The implemented capability system provides a set of versatile IPC primitives, which is the base of any microkernel architecture. These are based on the notion of a port, constituting an anchor to and from which partitions read and write messages. Ports allow system designers to specify the existing communication channels which can be characterized as synchronous or asynchronous, which trade-off security and performance. Asynchronous communication is safer since it does not require a partition to block, but entails some performance burden. In opposition, synchronous communication is more dangerous, since partitions may block indefinitely, but allows communicate to be faster, by integration with scheduler functionalities for efficient RPC communication.

One key feature of the μRTZVisor is that both client and server partitions use the same APIs, which homogenizes the semantics of communication primitives. The scheduling infrastructure also allows for the direct switch between guest client and server partitions, reducing overhead, and for secure tasks to be scheduled in their own right to perform background computations. This creates high-levels of

IPC traffic between the VM and the guest OS and higher number of context-switches but as the VMs are lightweight it significantly reduces overhead.

The real-time scheduler structure is based on the notion of time domains that execute in a round-robin fashion and to which partitions are statically assigned. Independently of their domain, higher priority partitions may preempt the currently executing partition, so that event-driven partitions can handle events such as interrupts as quickly as possible. However, the budget allocated to these partitions must be chosen with care according to the frequency of the events, to not be exhausted, delaying the handling of the event until the next cycle. This implementation is enhanced with a time-slice donation scheme in which a client partition may explicitly donate its domain's bandwidth to the target server until it responds, following an RPC pattern.

Finally, it is worth mentioning that the design and implementation of the μRTZVisor to a Zynq-7000 SoC following a single-core configuration and is heavily dependent on the implementation of TrustZone features on the platform.

3.1.1 Partition Manager

The partition manager is the kernel module responsible to keep track on the execution context of each partition, guaranteeing the integrity and consistency of their CPU state. The module keeps an array of all the Partition Control Blocks (PCBs), which contain information about each partition state as well as the partition that is currently active. The partition manager is used by other kernel modules to access the partition's PCB entries they are using.

As μRTZVisor supports two types of partitions the PCBs are different for each type, containing specific informations alongside common fields. The common information for both types that the PCB contains, is information about the capability space, communication channels, events, devices and the state of execution of each partition. The specific information kept tasks is limited to user mode CPU registers, in contrast to guest partitions in which are kept banked registers for all execution modes and co-processor state.

The partition manager, alongside keeping the execution state of each partition, is responsible for saving and restoring the context related to the processor state and coordinates the context-switching process among the different hypervisor core modules. When a different partition is scheduled, the partition manager is informed of the context change and the performs the context switch before leaving the kernel. As a coordinator of the system, it performs tasks such as invoking the

memory manager to switch be between address spaces when saving and restoring a partition state.

Other feature of the partition manager is that it can be used to delivery asynchronous notifications to task partitions, in a similar way to Unix-style signals.

3.1.2 Capability Manager

As mentioned before, the µRTZVisor implements a capability system to address partition access to system resources. The capability manager is the kernel module that manages the capabilities providing a fine-grained and flexible supervision of the resources. Each capability is an object, that keeps informations in a data structure about its owner, the object to which it is referenced and the permissions inherent to the capability itself. In the implemented system, every hypercall to the kernel requires the guest or task to provide the corresponding capability.

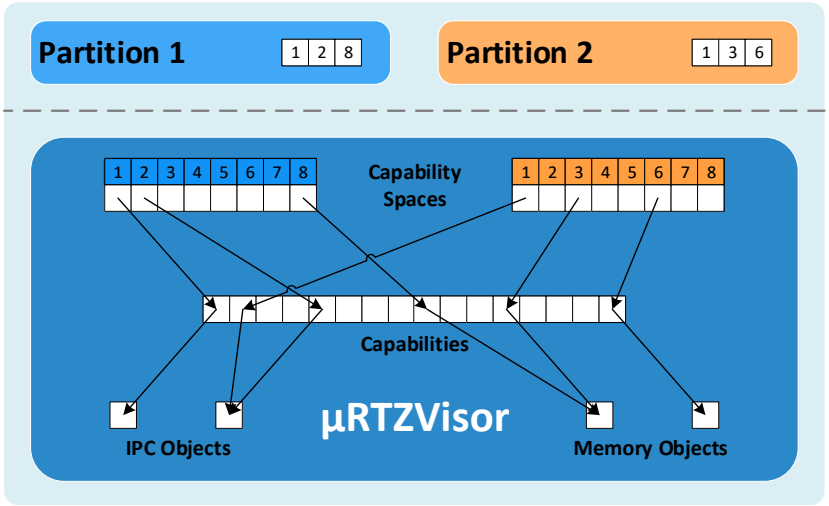


Figure 3.2: µRTZVisor Capability Sistem.

As capabilities are created and assigned to the partitions at initialization time according to the system design, the system guarantees that one partition does not interact with objects that it doesn't possess a capability to. There are fixed capabilities, which are always in the same position of the capability space of all partitions, as in example the capabilities that refer to objects as the address space. Other capabilities are given a name at initialization, name who's partitions need to fetch during execution to access the objects. Figure 3.2 gives a briefly illustration of the system. As an example, Partition 1 is configured with access to capabilities

1, 2 and 8 at design time, and if the partition desires to use the IPC, it needs, after initialization, to fetch the capability connected to the index 1 that assigned to a certain IPC object, in order to be able to send or receive messages from other partitions.

There is also the mechanisms to dynamic grant the access rights to other partitions through the IPC. The Grant operation consists of creating a derived capability, which is a copy of the original one, with only a subset of its permissions, and assigning it to another partition. This means that the granter must possess a capability for a port owned by the recipient. In turn, the revoke operation withdraws a given capability from its owner, and can only be performed by one of the partitions in a preceding grant chain. There is also a third type of capabilities which is called a one-time capability, that can only be used once. The first time a partition uses this capability it is erased from the partition's capability space.

3.1.3 Memory Manager

The Memory Manager module is responsible to do the building of all the address spaces for the partitions figuring out their layout in the physical memory. Since in Zynq-based devices, TrustZone memory segments have a granularity of 64 MB, the manager needs to check each partition memory space in order ot guarantee that when a guest partition is active, its the memory segment is configured as non-secure. This is required because if the guest is unaligned with a 64MB segment or if it the guest OS binary is bigger than 64MB, two segments must be configured as non-secure. This early address space building also detects if two guests were built to run by sharing the same memory segment and the manager will halt the system, since the spatial isolation requirement cannot be guaranteed. From the remaining free memory, the memory manager will build the virtual address spaces for secure tasks.

In addition, in the current implementation, no more memory can be allocated by tasks after initialization, so partition binaries must contemplate, at compile-time, memory areas to be used as stack and heap, according to the expected system needs. This module also manages two page tables used by the secure interface of the Memory Management Unit (MMU). The first is a 1-to-1 mapping to physical memory and is used when a guest partition is currently active. The second is used when a task partition is active and is updated each time a new task is scheduled.

Since it is expected that secure service partitions do not need to occupy a large number of pages, only the individual page table entries are saved in a list structure. The extra-overhead of updating the table at each task context restore

was preferred to keeping a large and variable number of page tables and only switching the page table pointer, reducing the amount of memory used by the hypervisor. Despite the transparent view of the top physical address space, this is controlled by managing a set of three extra page tables that map the 4 KB peripheral and the TZPC and TZASC partitioning infrastructure that enables control over guest access to peripherals and shared slices.

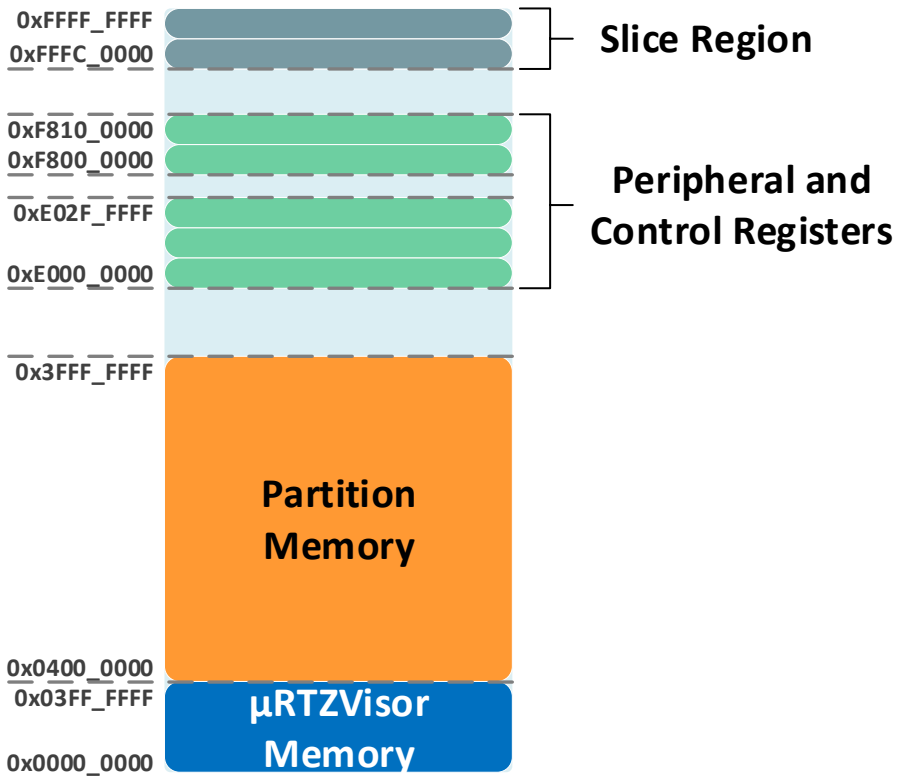


Figure 3.3: µRTZVisor Memory Configuration.

Figure 3.3 shows how the physical address space is organized. The hypervisor code and data are placed in the bottom 64 MB memory segment, which is always set as secure and mapped as kernel, privileged memory. The above memory region until the 1 GB limit is memory assigned to partitions. Above the 1 GB limit, the address space is fixed for all partitions, having the peripheral register area, and, at the top, a memory region of 4KB security configurable segments, which serve as slices and are used for guest shared memory.

Once the address space is determined, a capability is inserted into each partitions capability space that enables them to perform operations over it, such as creating and mapping objects representing some portion of their physical memory and that are support shared memory mechanisms. Two types of memory objects

are supported by the µRTZVisor being them page and slice objects, always represented and manipulated through capabilities and this objects can be granted through the IPC mechanism for usage by other partitions. Although both kinds of objects may be created by guest and task partitions, only slice objects may be mapped by guests, since guest address space control is exclusively performed through TrustZone segmentation mechanisms.

3.1.4 Device Manager

The Device Manager job is to manage a set of three page-tables and configuring TrustZone registers to enable peripheral access to task and guest partitions, respectively. Each peripheral comprises a 4 KB aligned memory segment, which enables mapping and unmapping of peripherals for tasks, since this is the finest-grained page size allowed by the MMU.

When a peripheral is assigned to a task, the entry for that device is modified to allow user mode access at each context switch. For all non attributed devices, the reverse operation is performed. An analogous operation is carried out for guests, but by setting and clearing the peripheral secure configuration bit in a TrustZone register. If a peripheral is assigned to a partition, it can be accessed directly, in a pass-through manner, without any intervention of the hypervisor.

At initialization, the device manager also distributes device capabilities for each assigned device according to the system configuration, in a similar form of the memory manager, as the peripherals are memory mapped. Here, when inserting the capability in a partition's capability space, the manager automatically maps the peripheral for that partition, allowing the partition to directly interact with the peripheral without ever using the capability. Partitions can also grant capabilities to use devices to other partitions. Another feature of the implementation is that the configuration that devices posses can also be secure and non-secure, which will influence the allowance of access to the DMA mechanism.

3.1.5 IPC Manager

The IPC Manager implemented on the µRTZVisor is based on the notion of ports, which are kernel objects that act as endpoints through which information is read from and sent to in the form of messages. Communication mechanisms are built around the capability-system, in order to promote a secure design and enforce the principle of least authority.

Similarly to devices and interrupts, ports are created at design time, and for a partition to perform a IPC operation over it, the partition must have a capability referencing the port with permissions for the operation wanted. In order to ensure the communication process between partitions each endpoint of a port should possess capabilities with the minimum read and write permissions. Port operations may work in a synchronous or asynchronous style, and are further classified as blocking or non-blocking.

Synchronous communication requires that at least one partition blocks waiting for another partition to perform the complementary operation, while, in asynchronous communication, both partitions perform non-blocking operations, but asynchronous communication requires a double data copy: first from the sender's address space to the kernel, and then from the kernel to the recipient's address space, being the with the lowest performance oriented operation.

Port Operations	**Sync**	**Async**	**Blocking**	**Non-Blocking**
Send	x	x	-	x
RecvUnblock	-	x	-	x
RecvBlock	x	-	x	-
SendReceive	x/-	x/x	-	x
SendReceiveDonate	x/x	x/-	x	-
ReceiveDonate	x	-	x	-

Table 3.1: IPC operations available over ports.

Table 3.1 summarizes all IPC primitives over available ports. As shown in the table, there are two kinds of receive operations — blocking and non-blocking. In blocking operations, the partition will wait for a complementary send operation to happen on the respective port to resume its execution. On non-blocking the partition will check for messages in the port's message buffer, which stores messages in FIFO fashion, returning one available message or an error value if the port is empty. When performing an operation with the -Donate suffix, the partition is donating its execution time-slice to the recipient partition, and it blocks its execution until receiving a response message from that same partition.

3.1.6 Scheduler

The Scheduler in the µRTZVisor was implemented in order to provide fast interaction between partitions as well as enabling the coexistence of both real-time and non-real-time applications. The main idea behind the architecture is to an execution window with a configurable, constant and guaranteed bandwidth in a round-robin style scheduler, which schedules time domains.

At design, each time domain is assigned an execution budget and a single partition. The sum of all execution budgets constitutes a execution cycle, which whenever a cycle is completed all the budgets are restored according to the design. The idea is that a partition executes in a time domain, until its execution budget is met, and then the scheduler will schedule the next domain. The implementation guarantees that each partition meets its execution budget in every execution cycle, providing a safe execution environment for time-driven real-time partitions.

Aside from the normal time domains, a special time domain is also provided, the domain-0, allows that to have assigned multiple partitions that are scheduled in according to the priority set at design time, in a time-sliced manner. At every scheduling point, the priorities of the currently active time domain's partition and the domain-0's highest priority ready partition are compared, and if the highest ready domain-0 partition priority has indeed an higher priority than the current domain, the domain-0 pre-empts the current domain, and starts to consume the domain-0 time budget.

Aside from the normal scheduling features, with the objective of having fast interaction between partitions, some of the functionalities were coupled with IPC operations. The ReceiveBlocking, SendReplyDonate and ReceiveDonate operations result in changes to the scheduling routine. The ReceiveBlocking operation allows to change the partition state to blocked, and then scheduling the next ready partition from domain-0 to do background work. In this case the domain-0 partition keeps consuming the blocked partition, as the running time domain is not changed. In this scenario the next scheduling point will be triggered in three cases. First case occurs when the domain-0 internal time slice expires resulting in scheduling the next highest priority partition from domain-0. The second case happens when the active time domain budget expires, and the next time domain is scheduled. The last case is when the executing partition sends a message to the active time domain partition, resulting in the unblocking of the previously blocked partition and scheduling it right away. Upon blocking, a partition remains in this state until it is unblocked by receiving a message on the port it is hanging.

The other operations, namely the SendReplyDonate and ReceiveDonate, result in a partition explicitly donating its time budget to the recipient port's owner, staying blocked until the message recipient sends its response. In case the donator has a higher priority than the donatee server, the latter will inherit the former's priority, augmenting the chances of it to execute and to resolve the dependency faster. This enables services to be provided in a priority-based manner, maintaining the priority semantics of the requesting partitions. However, a donate

operation may be performed to or from a partition that is already part of a donation chain in a transitive manner, constituting a more intricate scenario. Whatever partition is at the tail of the chain, it will be the one to execute whenever one of the preceding partitions is picked by the scheduler. As so a mechanism to detect deadlocks is provided, which aimed at being as lightweight as possible, considering its pervasive execution in every donation procedure.

One consideration in the implementation is that all the design was devised so that guest partitions are placed in common time-domains and secure task partitions are placed in domain-0, allowing to have the secure tasks encapsulating the kernel services, that can be configured with lower priorities executing only when they are useful for guest partitions. This model allows for the coexistence of event-driven and background partitions in domain-0, while supporting guests with real-time needs and that require a guaranteed execution bandwidth.

3.1.7 Interrupt Manager

As with any TrustZone-assisted system the GIC allows an interrupt to be configured as secure or non-secure, as well as supporting secure interrupts to be configured with higher priorities than non-secure. The CPU and GIC can also be configured so that all secure interrupts are received in monitor mode by the hypervisor as Fast Interrupt Request (FIQ) exceptions, and all non-secure interrupts to be directly forwarded to the non-secure world as Interrupt Request (IRQs). All of these features enable the hypervisor to have complete control over interrupts, their priority and preemption, while enabling pass-through access of guests to the GIC.

The implemented interrupt manager assumes that one interruption can only have one implemented handler in the entire system, regardless of the position in the system stack, as they may be in the kernel or in one of the partitions. If the interrupt is assigned to one partition in the configuration, the capability for the interrupt will be added to its capability space with the grant permission cleared. Partition interrupts are always initially configured as disabled and configured with the lowest priority possible.

Task partitions cannot be granted access to the GIC, since as they run on the secure world, they would have complete control over all interrupts. All interactions with the GIC are based on the hypervisor by invoking capability operations which allow tasks such as interrupt enable or disable. In contrast guest partitions can directly interact with the GIC, as a virtual GIC is maintained in the virtual machine.

(a) Normal Interrupt Handling. **(b)** Critical Interrupt Handling.

Figure 3.4: Interrupt Handling in the µRTZVisor.

While a guest is inactive, its interrupts are kept secure but disabled. Before running the guest, the hypervisor will restore the guest's last interrupt configurations as well as a number of other GIC registers that are banked between worlds and may be fully controlled by the guest. Active guests receive their interrupts transparently and as soon as they appear. Otherwise, as soon as the guest becomes active, the interrupts that became pending during its inactive state are automatically triggered and are received normally through the hardware exception facilities in the non-secure world. This can be seen in Figure 3.4a. If the partition has sufficient priority, it will preempt the currently active partition and immediately receive and handle the interrupt Figure 3.4b.

3.2 The Zynq-7000 SoC

As µRTZVisor was tailored for the Zynq-7000 SoC, the board chosen to develop the thesis was the low cost ZYBO board (Figure 3.5) which still manages to keep the processing power and extensibility of the Zynq Xilinx All Programmable System-on-Chip (AP SoC) architecture. The ZYBO [Dig16] is an entry-level platform suitable for embedded systems development built around the smallest member of the Xilinx Zynq-7000 family. The Zynq AP SoC is divided into two distinct subsystems, the PS and the PL.

The PS consists of many components, including the Application Processing Unit (APU) which includes two Cortex-A9 processors, an Advanced Microcontroller Bus Architecture (AMBA) Interconnect, a DDR3 Memory controller, and various peripheral controllers with their inputs and outputs multiplexed to 54 dedicated pins, also called Multiplexed Input/Output (MIO) pins. The ZYBO contains the Xilinx Zynq-7000 (XC7Z010-1CLG400C) SoC, which features a total of 28,000 logic cells, 240 KB Block RAM, 80 DSP slices, 12-bit On-chip dual channel Analog-to-Digital converter (XADC) on the PL side, which must be configured

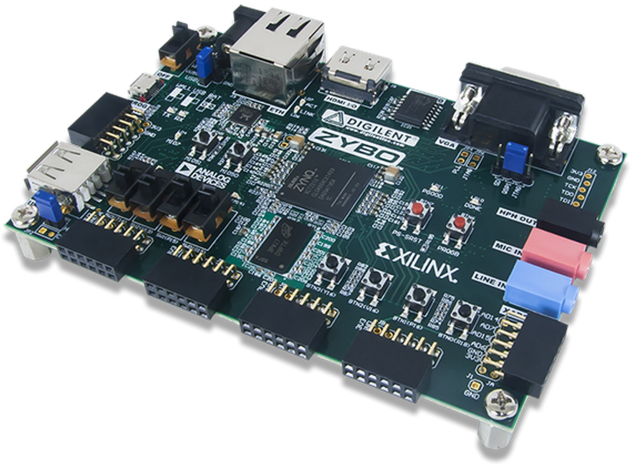


Figure 3.5: ZYBO Zynq-7000 Development Board.

either directly by the processor or via the JTAG port. The SoC also contains several low and high bandwidth peripheral controllers such as SPI, UART, I2C, 1G Ethernet, USB 2.0 and SDIO, and other ports like HDMI, VGA, microSD, Audio codec with headphone out, microphone and line in jacks. The overall SoC overview can be seen in Figure 3.6.

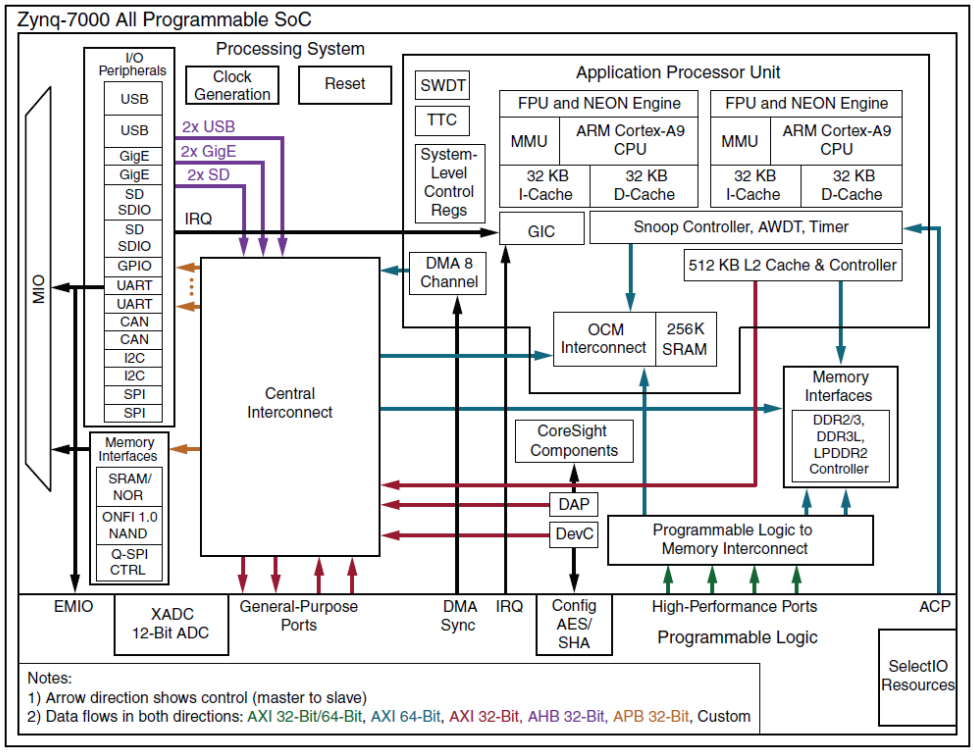


Figure 3.6: Zynq AP SoC architecture.

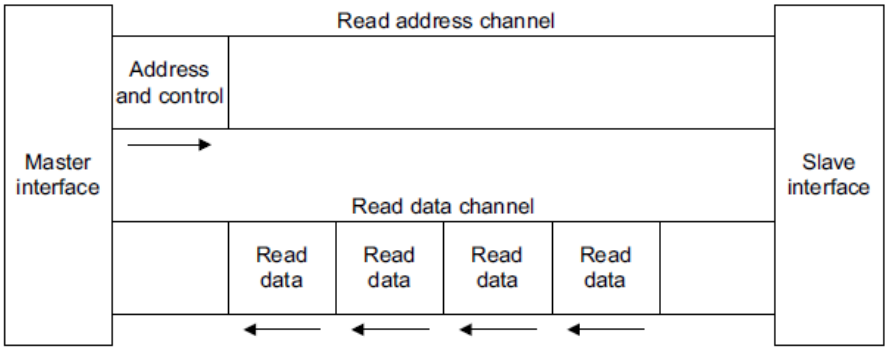
3.2.1 PS/PL Communication

The Zynq-7000 device builds up the communication bridge between PS and PL by implementing several types of interfaces which are based on a standard AXI protocol. Peripheral controllers that do not have their inputs and outputs connected to MIO pins can instead route their I/O through the PL, via the Extended-MIO (EMIO) interface. The peripheral controllers are connected to the processors as slaves via the AMBA interconnect, and contain readable/writeable control registers that are mapped in the processors' address space. The programmable logic is also connected to the interconnect as a slave, and designs can implement multiple cores in the FPGA fabric each containing addressable control registers. Furthermore, cores implemented in the PL can trigger interrupts to the processors and perform DMA accesses to DDR3 memory.

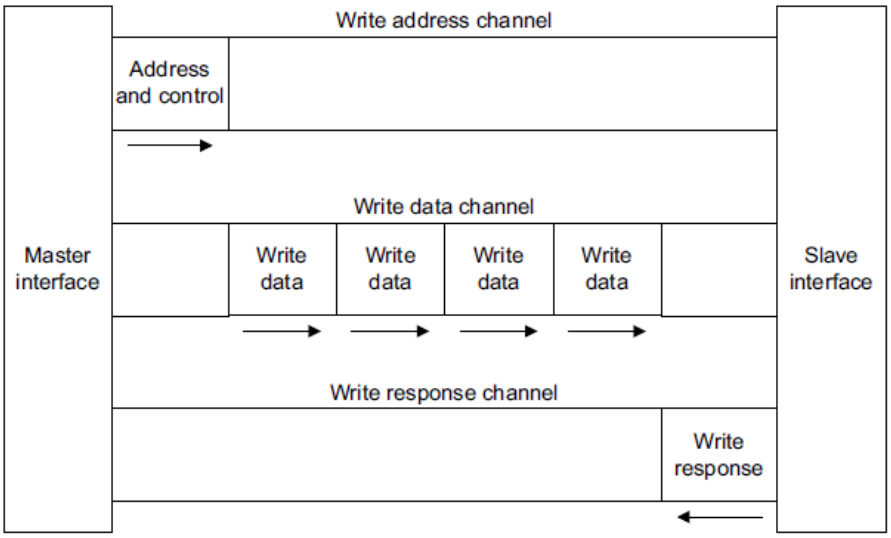
3.2.1.1 AXI

The Advanced eXtensible Interface (AXI) is the third generation of the AMBA, which is an interconnect specification that describes the connection and management of functional blocks in a SoC design. It was developed to support high-performance, high-frequency system designs, having separate phases regarding address/control configurations and data transmission on different channels, also using burst-based transitions, that allow write or read operations on sequentially addressed locations only issuing the first/start address. The AXI protocol includes the AXI4-Lite specification, with simpler control register style interfaces within, only allowing one data transfer per transaction, instead of the 256 data transfers in one burst allowed by AXI4 [ARM11].

As aforementioned, the AXI protocol defines a set of channels regarding transactions in the realm of control configurations and data transmission. These channels are: read address, write address, read data, write data and write response. An address channel carries the information that describes the nature of the data to be transferred. This data is then transferred between a master and a slave, through a read or write channel, depending on the transaction in hand. In a read transaction (Figure 3.7a), after the master indicates which is the address it is trying to read, data will flow from the slave to the master, through a read data channel. In a write transaction (Figure 3.7b), data will be transferred from the master to the slave through a write data channel and, after the data is written, the slave uses the write response channel to signal that the transfer has ended.



(a) Read Transaction.



(b) Write Transaction.

Figure 3.7: AXI Channel Transaction Flow.

The way in which AXI4 is organized, providing separate channels for addresses and data for both read and write transactions, allows for simultaneous bidirectional transactions, while enabling address information to be issued prior to the actual data transfer, which can speed up the communication process, given that the master and slave implement the necessary logic.

Each of the channels consists of a set of information signals and VALID and READY signals that provide a two-way handshake mechanism. The information source uses the VALID signal to indicate when valid address, data or control information is available on the channel. The destination uses the READY signal to denote when it is ready to accept the information. Both the read data channel and the write data channel also include a LAST signal to indicate when the final data item in a transaction is being transferred. The timing of this handshake mechanism, implemented in a few different fashions.

In each port there is an individual address channel for read and write transactions, and each appropriate address channel carries all the required address and

control information for a transaction. The read or write data channel carries the read or write data and the read or write response information from the slave to the master, while supporting a data bus, that can be 8, 16, 32, 64, 128, 256, 512, or 1024 bits wide depending on the burst size. Write data channel information is always treated as buffered, so that the master can perform write transactions without slave acknowledgement of previous write transactions.

The majority of systems with AXI consist of several master and slave devices connected through an interconnect, providing a single interface definition between a master and the interconnect, a slave and the interconnect and a master and slave.

PL Mode	Port	Throughput	Description
Slave	M_AXI_GP0 M_AXI_GP1	600 MB/s	General Purpose Ports
Master	S_AXI_GP0 S_AXI_GP1		
	S_AXI_ACP	1200 MB/s	Accelerator Coherency Port
	S_AXI_HP0 S_AXI_HP1 S_AXI_HP2 S_AXI_HP3	1200 MB/s	High Performance Ports

Table 3.2: AXI Ports on the Zynq-7000 SoC.

All the AXI port present on the Zynq are shown in Table 3.2. The AXI GP enable the PS to access the PL directly is through the interface with the reserved memory address space associated with it. Within this range the user can map several PL modules and interact with them. Contrary to the AXI-HP and AXI-ACP interfaces which only support slave mode in the PS, the AXI-GP support both master and slave modes.

3.2.2 Partial Reconfiguration on Zynq

On Xilinx based systems, a PR system allows to have multiple user-defined logical sections can be dynamically reconfigured. Each logical section is a RP with the ability to have unlimited implemented configurations inside. For every Reconfigurable Module (RM) for each RP, a partial bitstream is generated.

Partial bitstream files are processed just like normal bitstreams on a full reconfiguration of the FPGA, and contain all the configuration commands and data necessary for a reconfigurable partition and the implemented logic. They have the

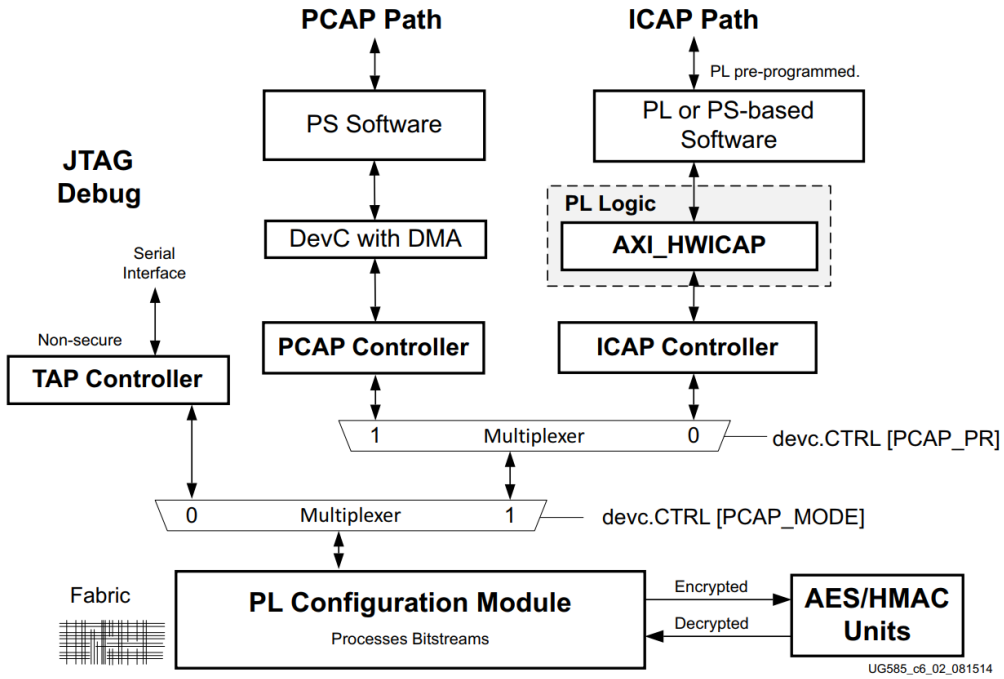


Figure 3.8: Reconfiguration Mechanism Selector.

characteristic that they cannot be sent to the wrong logic area, as they contain the configuration for specific hardware components. As such bitstream file sizes will vary only depending on region size and resource type contained in the region.

On Zynq-7000 AP SoCs, PR can be performed using either the PCAP or the ICAP and additionally the JTAG interface, the last one being more suitable for debug purposes. All the interfaces are mutually exclusive and cannot be used simultaneously, but there is the possibility to switch between in runtime using the DevCfg registers as shown in Figure 3.8. The reconfiguration times will depend on two factors, the configuration bandwidth of the used port and the partial bitstream file size.

3.2.2.1 PCAP

The Processor Configuration Access Port (PCAP) is a reconfiguration controller used for PR. The PCAP is accessed through a Device Configuration Interface (DevCfg) that has a dedicated DMA controller to transfer the bitstreams from the DRAM memory to the PCAP in order to have the reconfiguration executed. The PCAP is a configuration interface that is tightly coupled with the PS region of the SoC and has the capability of performing partial and full reconfigurations. The port allows to write and read configurations, but is limited to a theoretical maximum throughput of 145MB/s.

3.2.2.2 ICAP

The Internal Configuration Access Port (ICAP) also provides the ability to perform PR but as it requires the instantiation of an ICAP controller as well as logic to drive the bitstream to the ICAP interface on the FPGA, it cannot perform full reconfigurations. Additional to the configuration interfaces being mutually exclusive, the ICAP is only available after the FPGA is configured and can only be accessed once the other ports are de-selected.

This option requires more engineering effort but is useful when higher performance is needed with configuration bandwidths up to the theoretical 400 MB/s.

3.2.3 AXI Direct Memory Access

The AXI Direct Memory Access (AXI DMA) IP core [Xil18a] provided by Xilinx allows for high-bandwidth transfers between the AXI4 memory mapped and AXI4-Stream IP interfaces. Another feature that the IP has is that it can offload data movement tasks from the CPU in processor-based systems, with the Scatter/Gather Engine it possesses. The IP core is accessible through the AXI-Lite interface and enables the PS to configure and manage the core registers as well as starting transfers. Figure 3.9 illustrates the overall block diagram of the AXI DMA core.

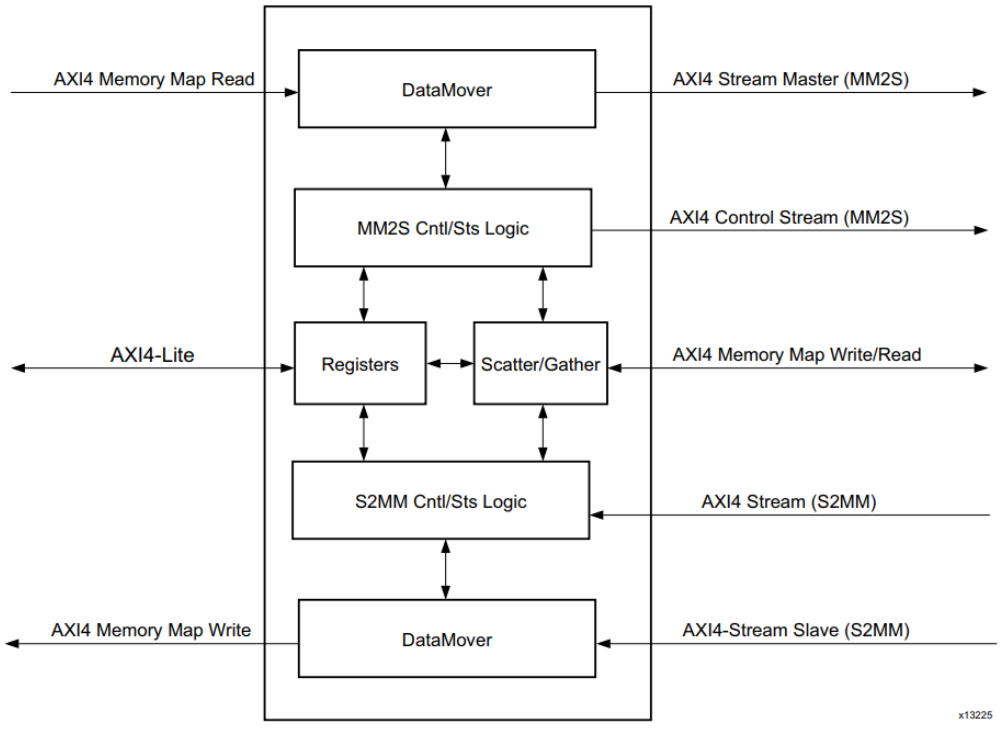


Figure 3.9: AXI DMA Block Design.

Primary high-speed DMA data movement between system memory and the target is done in stream type transfers through the AXI Read Master to AXI DMA Memory-Mapped to Stream (MM2S) Master and AXI Stream to Memory-Mapped (S2MM) Slave to AXI Write Master. AXI DMA also enables up to 16 multiple channels of data movement on both MM2S and S2MM paths in scatter/gather mode. An important characteristic of the IP is the ability of both MM2S channel and S2MM channel operating independently.

If the Scatter/Gather Engine is excluded from the IP through configurations, a less FPGA resource intensive mode core can be enabled. In this mode transfers are commanded by setting a Source Address for MM2S or Destination Address for S2MM and then specifying a byte count in a length register.

Channel	**Clock** (MHz)	**Bytes Transferred**	**Throughput** (MB/s)	**Percent of Theoretical**
MM2S	100	10000	399.84	99.76
S2MM	100	10000	298.59	74.64

Table 3.3: AXI DMA channels measured Throughput.

Table 3.3 shows the throughput obtained in a test presented in the reference manual, transferring 10000 bytes through the AXI DMA without using the Scatter/Gatter engine in both directions. The test revealed the MM2S to be a good candidate to perform the transfers from the memory to an ICAP interface as it is limited to 400MB/s.

4. µRTZVisor DPR Framework

In this chapter, the mechanisms implemented to support the reconfiguration of the RMs and their interactions with the multiple guest OSes and the PS will be described in detail. The explanation of the most relevant design options taken as well as the reasons behind them will be presented before showing the modifications required to integrate the mechanism in the hypervisor.

4.1 Overview

In order to achieve a system where the hypervisor is able to dynamically manage software and reconfigurable hardware tasks, several components were implemented towards creating a DPR framework that allows such flexibility without deteriorating the overall performance significantly. Being a software-hardware co-designed solution, implemented to support concurrent reconfiguration requests of multiple guest OSes on demand, it was necessary to integrate components in both the software and hardware layers.

The software component of the framework that in a monolithic architecture would be placed within the kernel, is implemented as a secure task, relying on the provided architecture of µRTZVisor, as a microkernel-like hypervisor. While still being a relevant feature of the hypervisor, the service fits the characteristic concept of keeping the kernel as simple as possible, with the complexity of the kernel remaining unmodified.

Having in mind the necessity to implement a mechanism that prevents the jeopardizing of the hardware modules, the service acknowledges and gives proper answers to the multiple requests while being the only one allowed to trigger the reconfiguration, as it is the only partition in the hypervisor configured with the capabilities to have access to the reconfiguration mechanism. The communication mechanism between the guests and the service is established through the IPC, and any guest that wishes to use a hardware accelerator can send requests, as long as the ports between the partitions were created and configured. As mentioned in

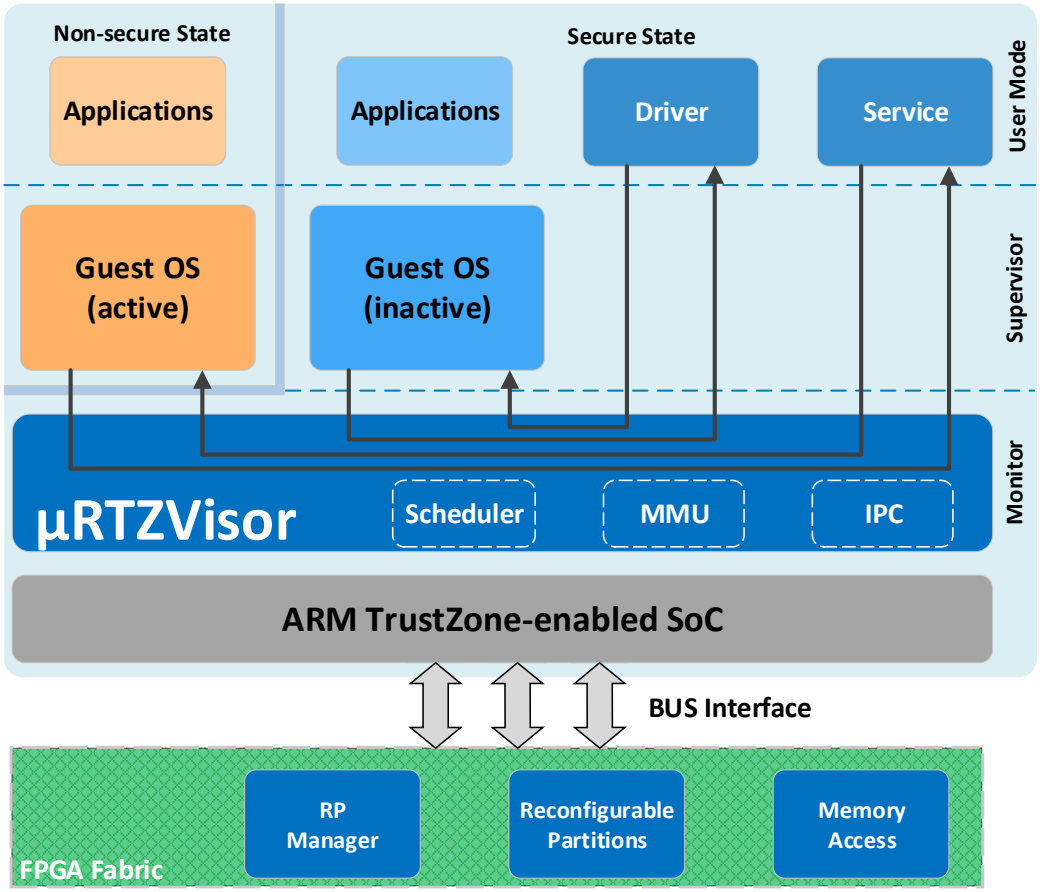


Figure 4.1: µRTZVisor focused DPR Architecture Overview.

the previous chapter, ports are established at design time and are used to allow the communication between partitions. If there are no ports bridging the guest partition and the service, the guest cannot use the functionality.

To execute the reconfigurations, the system was implemented using the ICAP configuration port, enhanced with AXI DMA based bitstreams transfers, which allows for reconfigurations with theoretical maximum throughput up to 400 MB/s. On the hardware side, alongside the ICAP interface to accommodate the reconfiguration of the RMs, multiple RPs are placed and connected to the two other main modules, a controller and a module for memory interactions. An overview of the proposed architecture can be seen on the Figure 4.1, which depicts the interfacing of the guests through the IPC to the secure task, and the overview of the hardware.

Following the concept behind the architecture, it is possible to identify five main components:

- **Reconfiguration Mechanism** - using the ICAP coupled with AXI DMA

transfers, while verifying its performance and making it capable of reconfiguring multiple modules;

- **Reconfigurable Partition Manager** - hardware module that serves as a synchronization mechanism to the service implemented as a secure task while managing all the modules implemented on the hardware. It manages and stores information about the current configuration of the RM present in each RP. It also grants and establishes the connection between the guest data and the RMs underutilization. Additionally, the manager interfaces the module with the reconfiguration mechanism to control the reconfiguration flow, blocking unwanted reconfigurations and misbehaviours;

- **Memory Access** - hardware module that is capable of accessing the DRAM using the AXI HP ports for maximum throughput. Analogously to the RP Manager, it is connected to every RP, and supports burst read and write operation requests from each module;

- **Reconfigurable Partitions** - containers for RMs developed with a common interface that can be used to accommodate any type of function only being limited to the hardware resources it contains. For example, if the RP has multiplication functions that require DSP functionalities it needs to contain at least one DSP slice on the assigned FPGA portion. Common interfaces and execution flow allow for the same functionality to be implemented in several RPs;

- **Software Service and API** - software component of the architecture that is responsible to respond to the requests from the various guest OSes. Alongside the service, a guest-service API defining a communication policy will be established to make the request handling as efficient as possible.

Having the components and their implications to the system established, the system was implemented in an iterative way starting from the reconfiguration mechanism and progressing to the modules and their management and finalizing with the software interface with the guest OSes.

Another feature of the proposed architecture is to have interrupts that signal the end of execution of the hardware accelerators, allowing guests to avoid having the traditional pooling methods to detect if the requested hardware accelerator has finished its execution, allowing parallel processing between software and hardware. Each of the components will be explained in detail in further sections as well as the interfaces used to communicate with software.

4.2 Hardware Modules

Before we begin with details of the implementation a brief, but more elaborated view of the hardware, alongside with the interfaces used between the PS and the PL, is introduced. Figure 4.2 depicts a simplified view of the implemented hardware showing the main modules present in the FPGA as well as the existing interfaces between each module and with the PS.

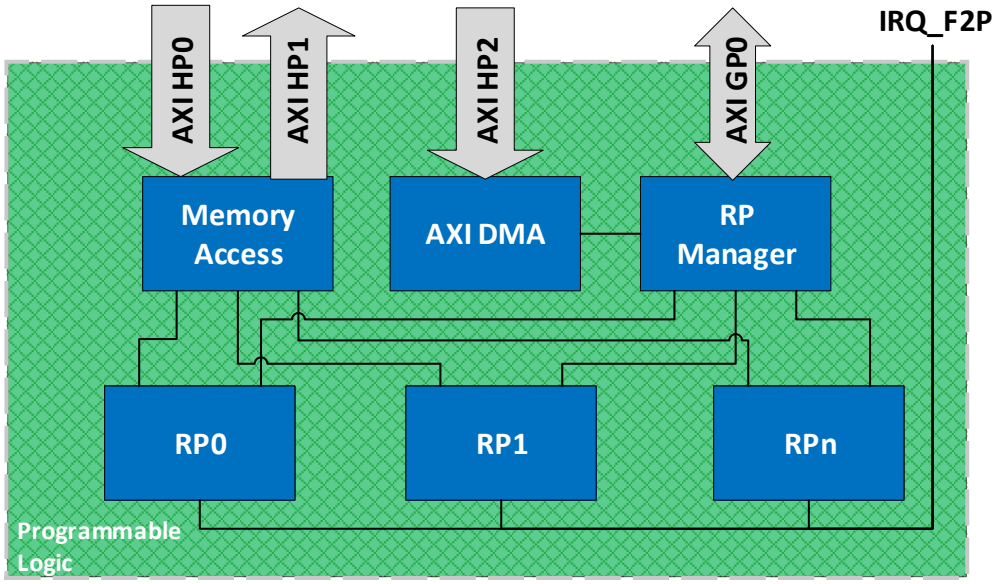


Figure 4.2: Implemented Hardware Overview.

As explained in the previous section, the design is constituted by three main hardware components interconnected between them and with the software. The PS to PL interfaces required were the following:

- **AXI HP** - two channels were used to support concurrent read and write operations to the DRAM where the guest data intended to be processed in the RMs is stored. The choice fell on the high-performance ports as they allow stream and burst based transfers for maximum throughput. One channel is used for memory reads and the other is memory writes working independently from each other.

- **AXI_DMA** - used to transfer the bitstreams from the DRAM to the hardware logic. The IP provided for Xilinx is connected to a third AXI HP port and uses stream type of transfers to transfer blocks of memory. It is very suitable for this application as it allows to transfer full bitstreams in just one transaction.

- **AXI GP** - used to exchange information between the RP Manager and the Secure Task.
- **IRQ_F2P** - One interrupt is used for each RP to signal the guest, to which it is currently assigned, that the module execution is complete. Another interrupt is connected to the ICAP interface, and signals the end of the reconfiguration process.

The following sections of this chapter will delve deeper into each component, explaining use cases and execution flow of relevant developed features.

4.2.1 Reconfigurable Partition Manager

Regarding the management of reconfigurable modules in this framework, the goal was providing an easy to scale, reliable and fast response system, capable of handling multiple RPs. The overview for the module is shown in 4.3. It is composed by three types of interfaces, namely the interface where it receives the bitstream from the AXI DMA in a stream type, the interface with the software task based on the AXI Lite registers which allow the interchange of information, and the interface with each RP. These interfaces are connected to two control units, one that handles the reconfigurations and another that gives appropriate responses to each request.

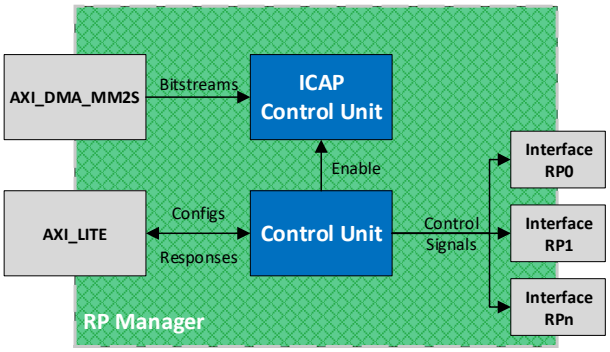


Figure 4.3: RP Manager Module Overview.

To meet the design feature of implementing a system capable of being easy to scale, the manager was implemented in a modular model, minimizing the engineering effort required to introduce or remove RPs in the system. The state machine shown in 4.4, is an overall overview of the states that the manager executes for a system with three RPs. To introduce a new RPn+1, n being the current number of RPs on the system, there is the need two new introduced states, one for

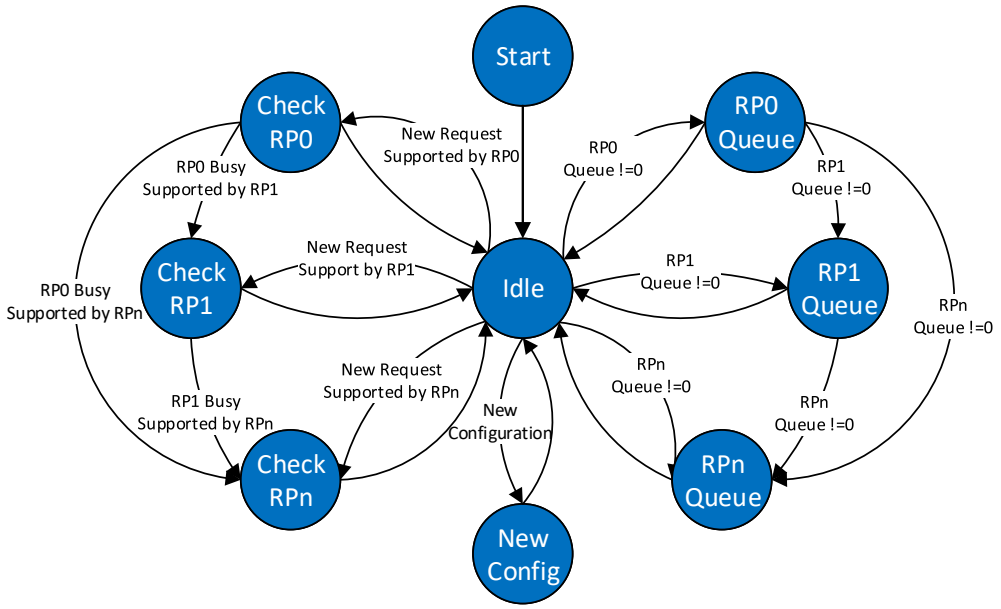


Figure 4.4: RP Manager State Machine.

checking the state of the RP when new requests arrive and another for the queues. Although the two states represent a significant addition in lines of code, they represent little to no engineering effort as they are essential copies of the RPn states. The modifications decay over how the transition between states is done. Worth mentioning that there is a state for when new configurations are added to the system and they are introduced in lists. They are used so the manager knows what RPs are compatible with the requested RM.

When a guest requests a hardware accelerator that is implemented as a RM, the software service sends the guest ID and the ID referent to the hardware accelerator to the RP Manager, since the response to the request is dependent on the current state of the hardware, via the AXI GP channel that is mapped to the RP Manager and allows to read and write registers from each side of the management unit. The request is then compared to the current state of the accelerators in each RP. From the comparison four cases can occur:

- An RP has the configuration correspondent to the request is in idle state, allowing the assignment to the guest to be done on demand, and the execution starting right away;
- An RP has the RM configured but is currently being used by another guest. In this case, and taking into consideration power consumption, if the reconfiguration time is longer than the remaining execution, the guest is introduced in a queue state, which is then assigned to the RP as soon as the

module ends its current execution;

- The configuration is not currently in any RP, but there are available partitions capable of holding the hardware accelerator, so the reconfiguration is made and the module is assigned to the guest. When the interrupt that signals when the ICAP has finished the reconfiguration process module occurs, the module starts executing its task;

- The last option is when any of the above cases cannot happen, and all the RPs are currently in use and their queues full, so the RP Manager returns a busy response to the software service and the guest is informed that the request cannot be fulfilled.

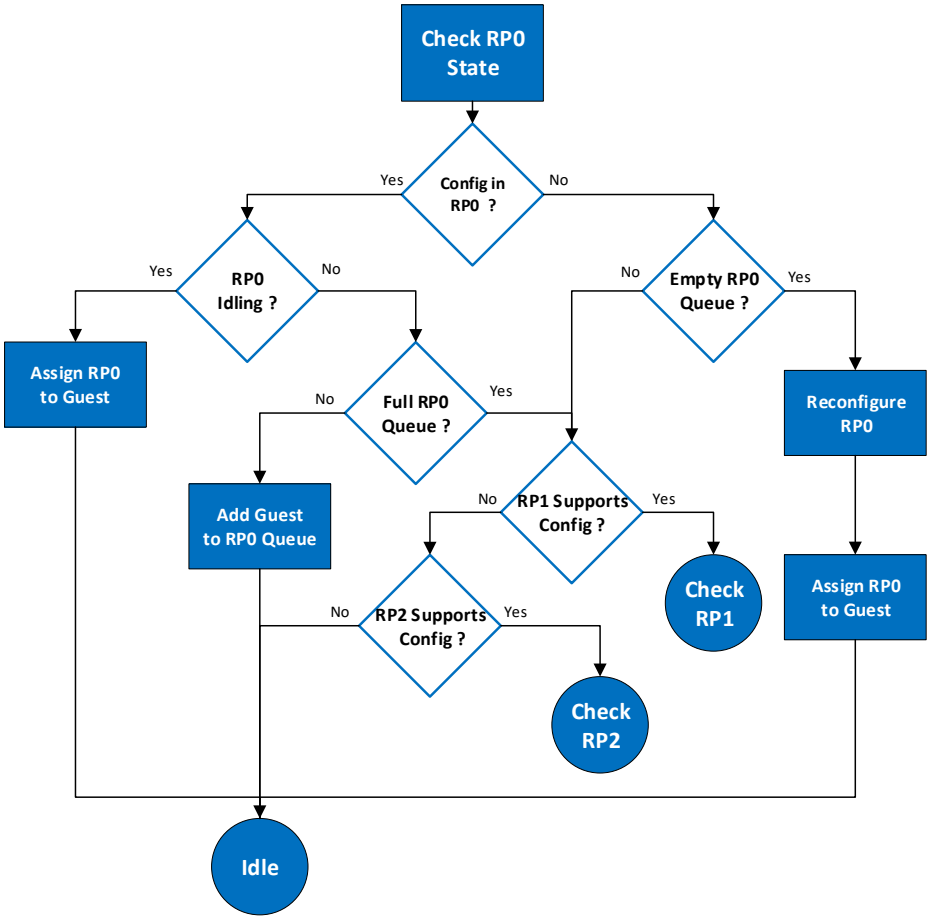


Figure 4.5: Check RP0 Execution Flow.

The flowchart presented in Figure 4.5 provides a better illustration of the process that happens when a request is received and how the queues are connected to the requests.

4.2.2 Reconfiguration Mechanism

To support the module reconfigurations there was the need to implement a hardware module capable of establishing the interface between the data stream resultant from the AXI DMA memory to stream channel to the ICAPE2 primitive that connects the hardware logic to the ICAP.

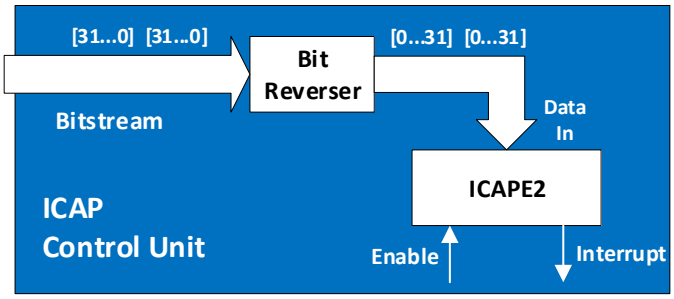


Figure 4.6: ICAP Control Unit Overview.

As Figure 4.6 shows, the control unit needs to reverse the bits from the word received coming in the input stream that te DMA provides. This is due to the requirement that the data input port from the ICAPE2 primitive has that needs to receive the least significant bit in the position of the most significant bit, so it requires the word to be mirrored. Additionally, the interface is enabled by the RP Manager control unit when reconfigurations are necessary, providing another safety level to the implementation. When the last frame is sent to the ICAP the interrupt is triggered.

4.2.3 Memory Access

The memory access module represents another essential component in the hardware, as every module that is present each RP is dependent on data stored in the DRAM, that belongs to the guest that desires to use the hardware accelerator. Again to sustain the design feature of an easy-to-scale system, the interface with the RPs is the same for all of them. Figure 4.7 depicts the overview of the Memory Access module, which is composed by an interface with each RP and interfaces with the AXI HP channels used to the read and write data transactions.

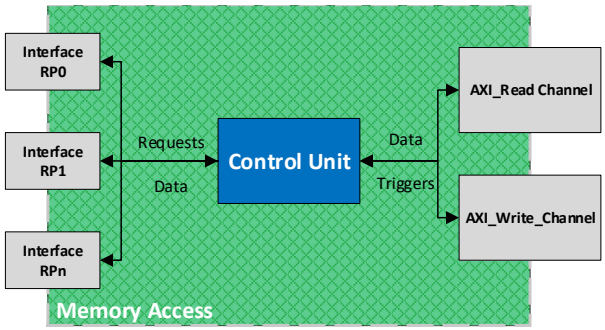


Figure 4.7: Memory Access Module Overview.

The flowchart presented in Figure 4.8 shows the behaviour of the control unit connected to the read channel, and how the module is constantly waiting for requests allowing the modules waiting time to be as small as possible when they request memory read operations. An identical control unit is implemented for the write requests, which is then connected to the write channel.

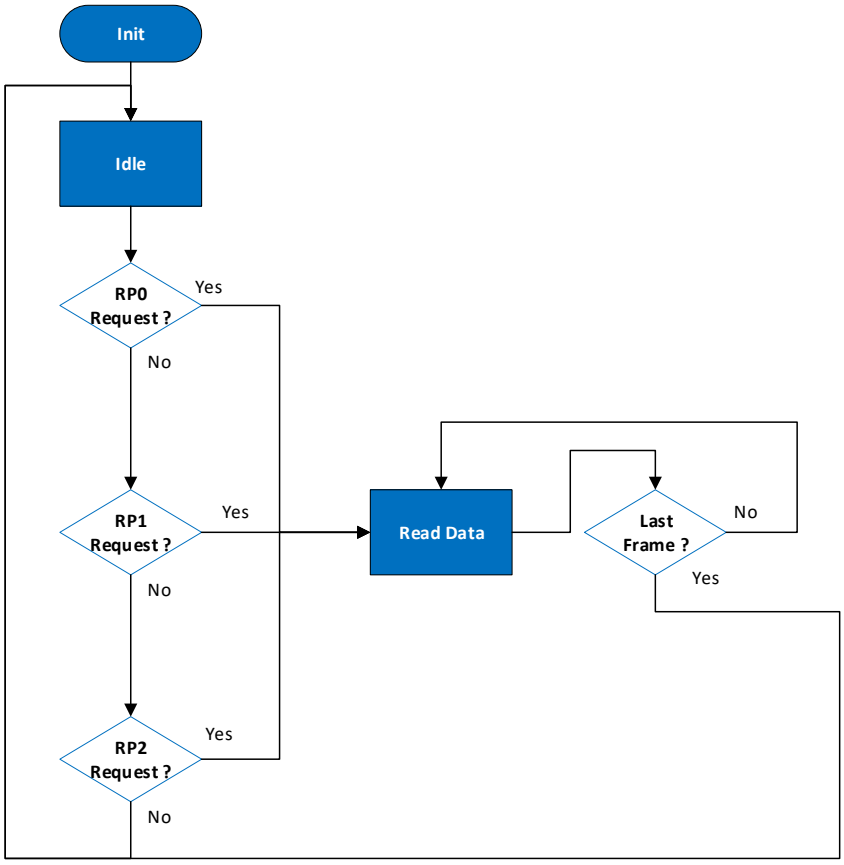


Figure 4.8: Memory Access Read Branch Execution Flow.

4.2.4 Reconfigurable Partitions

The reconfigurability is an attribute of a partition, which is set on the moment it is instantiated, and allows the implementation of multiple different RMs and the respective partial bitstream generation used to configure the RP dynamically during runtime. When designing an RP, it is essential that the unit is capable of holding the largest RM while also having all the ports necessary to interface the static logic with the RM. A good practise when designing RMs is implementing common interfaces to the RP, as partition pins are automatically created for all RP ports. If an RM uses different inputs or outputs from another RM, the resulting RM inputs or outputs might not be connected.

Following these recommendations, implemented RPs have the same set of ports, allowing RMs to be exchangeable between them, which then increases the versatility of the system. This enables different guests to be using the same hardware functionality in different RPs concurrently as well as making adding or removing RPs easy. The changes necessary to add an additional RP to the system is simply creating another port in both the RP Manager and Memory Access modules alongside the adaptations required to the state machines both modules are using.

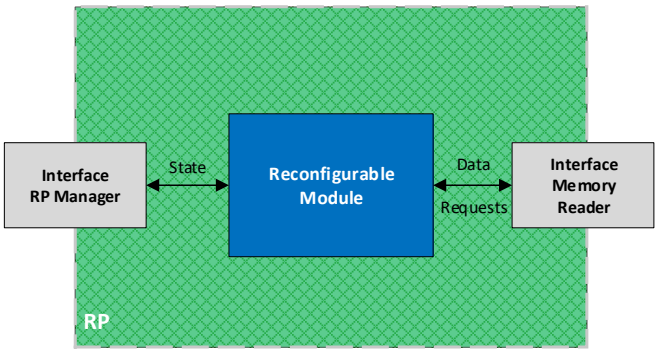


Figure 4.9: RP Overview.

The general execution flow is to have the hardware accelerator implemented receive the trigger to start executing, which then makes the module request to read the data stored in the memory, execute the processing of the task and then write the results on the DRAM, using the Memory Access module for the operations. The generic process is depicted on the Figure 4.10.

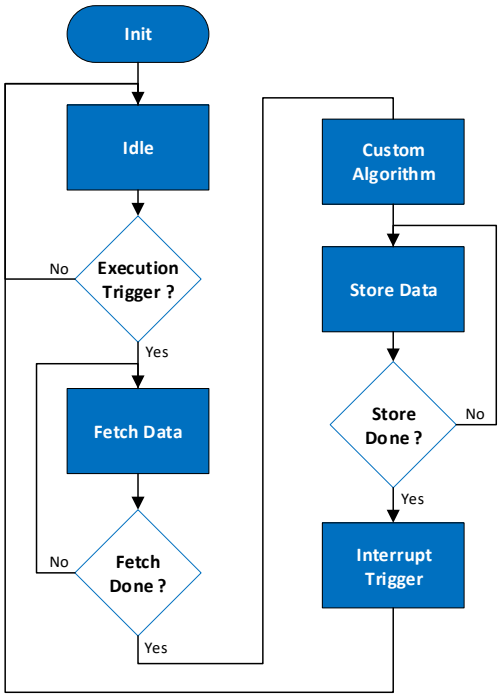


Figure 4.10: RM example Execution Flow.

As the data is fetched to the hardware through the Memory Access Module, the amount of data that can be processed in one execution cycle is directly related to the configuration of the AXI HP ports. As an example, if the port is transferring bursts of 32 words, then the RM will be able to process 32 words at a time. One module can requests multiple read and writes if necessary. Additionally, both the read and write functions are also optional, and can be bypassed if the implemented module does not require it.

4.3 Software Task

The software service was developed regarding the management of multiple RPs while keeping the design goals of providing an easy to scale, reliable and fast response system, thus the implementation and execution flow are highly affiliated to the RP Manager. The task is the only part of the PS that is capable of accessing the AXI-Lite connected to the hardware manager and also encompasses the interface between the guests OSes and the hardware. The AXI-Lite connected to the RP Manager contains the registers presented in table 4.1, and are used as the interface between the task and the manager. The response register returns information to the software according to the current state of the RPs and if a

reconfiguration is need to satisfy the request, the response is used to select the appropriated bitstream.

Offset	Type	Name	Description
0	W	GUEST_ID	ID of the Guest
4	W	GUEST_REQUEST	ID of the Configuration Requested
8	R	RESPONSE	Response
12	W	START_RPs	Start Trigger
16	W	NEW_CONFIG	Add new configuration ID
20	R	ACTIVE_GUESTS	Guests Currently Using the RPs
24	W	G0_RADDR	ADDR of the Data to be Processed
28	W	G0_WADDR	ADDR to write the Processed data
...	W	Gn_RADDR	ADDR of the Data to be Processed
...	W	Gn_WADDR	ADDR to write the Processed data

Table 4.1: RP Manager register map.

The developed API encompasses four functions which allow the guests to request for generic accelerators provided by the service itself or for custom private accelerators, and are the following:

- **Add_Private_Configuration()** allows the guest to inform the secure task responsible for the reconfiguration mechanism that it has custom private configurations for each RP. The API returns to the guest an ID that should be kept and is used to request the configuration;
- **Remove_Private_Configuration()** allows the guest to remove private configurations from the available configurations. For that the guest needs to use ids that were assigned in the previous API;
- **Request_Private_Configuration()** following the same line of thought the guest can request the usage of the accelerator by sending a message to the secure task with the ID of the private configuration.
- **Request_Configuration()** similar to requesting a private configuration, but for general configurations that the secure task provides.

It is worth mentioning that the guest needs to inform the task that it wants to use private configurations before requesting to use them. The execution flow for the service can be seen in Figure 4.11. The hypervisors' event manager is used to signal the task when the ports created between the task and the guests have

messages, and according to the message it executes a function to determine the appropriate response. Additionally, when a hardware task finishes execution it warns the guest who was having its data processed on the task that the data is ready.

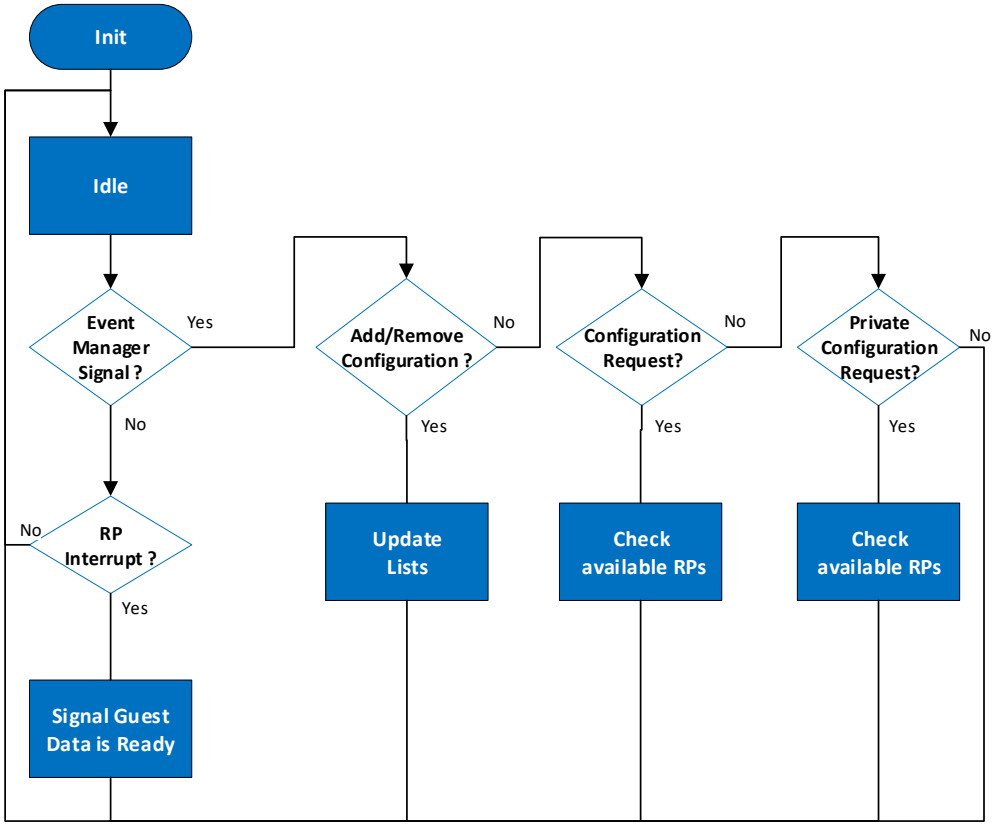


Figure 4.11: Software Service Execution flow.

When a guest requires to use a configuration, it can receive several responses from the secure task as aforementioned. Figure 4.12 shows the overall sequence that happens for each case in a request for a private configuration. When requesting a private configuration, the number of interactions with the secure task will increase in case a reconfiguration is necessary.

This is due to the fact that private guest configurations are stored in the memory space of the OSes, and even although the task is on the secure world, it requires a capability to access the memory space of the guest in which the bitstream is stored. So the guest needs to create a capability to the bitstream memory space and send it to the task using the IPC mechanism. When the task receives the capability it can then transfer the bitstream to the ICAP and reconfigure the hardware accelerator. The guest is then notified that the reconfiguration

is running and the data will be processed shortly after. This case is, however, the worst case scenario, and the most extensive interaction between a guest and the task.

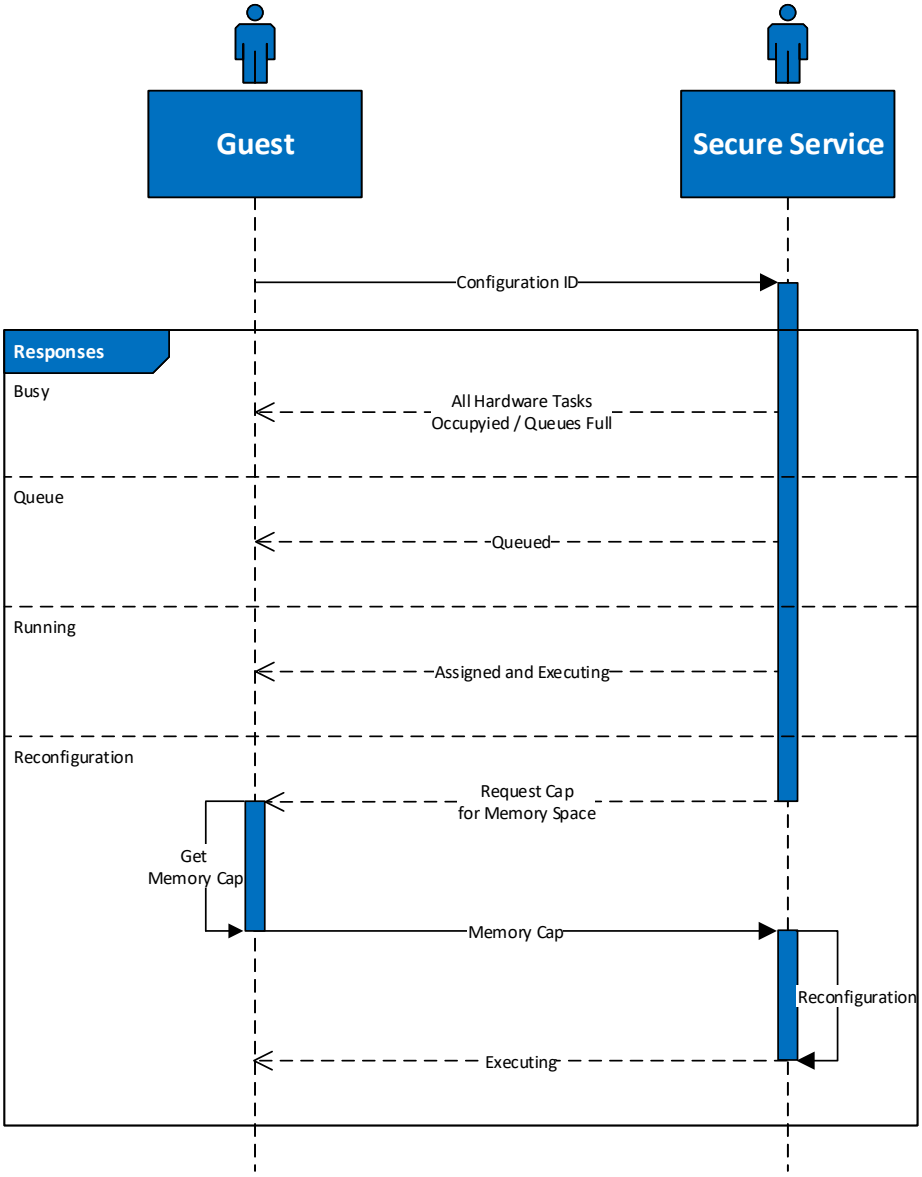


Figure 4.12: Private Configuration Request Execution flow.

When the reconfiguration is not required in private configuration requests, the response time of is the same as for requests for generic configurations.

4.4 µRTZVisor Integration

The last phase of the implementation was the integration of the complete framework in µRTZVisor which required several modifications to its current configuration, as well as some additional functions to be implemented in order to execute without problems. The modifications encompassed the configurations for the AXI DMA, DevCfg, AXI security settings, interrupts, and many others who will be explained in detail in this section.

The first modification was to modify the TrustZone security registers of both the DMA and AXI, seen in Listing 4.1. They were configured to always operate in secure state, as the active guest runs in non-secure mode and the only partition capable of accessing them is the Secure Task. This way, if the processing finishes after the guest stops executing and is on the secure world, the data can still be written to the guests memory assigned to the hardware task.

```
1 //Unlock SLCR register
2 write32(platform::SLCR_UNLOCK, platform::SLCR_UNLOCK_KEY);
3 //DMAC TrustZone Config: Secure State
4 write32(platform::TZ_DMA_NS, read32(platform::TZ_DMA_NS) &
      ~(1<<0));
5 //DMAC TrustZone Config for Interrupts: Secure State
6 write32(platform::TZ_DMA_IRQ_NS, read32(platform::TZ_DMA_IRQ_NS
      ) & ~(1<<0));
7 //DMAC TrustZone Config for Peripherals: Secure State
8 write32(platform::TZ_DMA_PERIPH_NS, read32(platform::
      TZ_DMA_PERIPH_NS) & ~(1<<0));
9 // Configure PL access: Always Secure
10 write32(platform::SECURITY_FSSW_S0, 0x0);
11 write32(platform::SECURITY_FSSW_S1, 0x0);
12 write32(platform::SECURITY_APB, 0x0);
13 // Configure Master peripherals mode: Always Secure
14 write32(platform::TZ_FPGA_M, 0x0);
15 write32(platform::TZ_FPGA_AFI, 0x0);
16 //Lock SLCR register
17 write32(platform::SLCR_LOCK, platform::SLCR_LOCK_KEY);
```

Listing 4.1: Configurations for secure AXI and DMA.

Further modifications were required to the device management of the hypervisor as the AXI DMA and AXI peripherals, both connected to the FPGA logic, which were previously not mapped in the page tables of the hypervisor. This

modification encompasses the mapping of the page tables for all the AXI based peripherals, Listing 4.2.

```
1 static constexpr uint32_t * const device_table3 = &
    device_table_3;
2 static const uint32_t table3_base = 0x40400000;
3 static constexpr uint32_t * const device_table4 = &
    device_table_4;
4 static const uint32_t table4_base = 0x43C00000;
```

Listing 4.2: Additional page tables for AXI peripherals.

The AXI DMA required another adjustment to the hypervisor device management since in the previous implementation each peripheral constituted a 4 KB aligned memory segment, which enables mapping and unmapping of peripherals for tasks, since this is the finest-grained page size allowed by the MMU, but the AXI DMA requires further pages, as it is a 64MB device. On context switch the configuration TrustZone registers changes to enable peripheral access to task and guest partitions, respectively, and for this the function was altered to accommodate the 16 pages for the device, as presented in Listing 4.3.

```
1 if(id == DEV_AXIDMA){
2   for(i = 0; i < 15; i++) {
3     uint32_t index = ((device_address_table[id].page_addr) & 0
        x000FFFFF) >> 12;
4     index = index + i;
5     if(active){
6       table_ptr[index] |=  0x20;
7     }
8     else {
9       table_ptr[index] &=  ~(0x20);
10      assembly::mmu_flush_tlb_mva(device_address_table[id].
          page_addr);
11    }
12  }
13 }
```

Listing 4.3: Device configuration for AXI DMA.

When guests create the capability that they send to the DPR task in order to reconfigure the hardware with private bitstreams, the memory capability has

the virtual address which is incompatible with the physical address that the AXI DMA requires to transfer the bitstream data to hardware. In order to allow that an hypercall was introduced that calls the memory manager to translate the address (Listing 4.4).

```
1 int32_t MemoryManager::Translate(uint32_t addrspaceref,
      AddressSpaceArgs* args){
2         int32_t ret = -1;
3         uint32_t addr = args->base;
4
5         if(TranslateVirtToPhys(addrspaceref, &addr)){
6                 args->base = addr;
7                 ret = 0;
8         }
9         return ret;
10 }
```

Listing 4.4: Translate Function added to the Memory Manager.

Once all the modifications were done to µRTZVisor, the focus was to do the proper configurations in order to have the task communicating with the hardware and using the reconfiguration mechanism. The modifications involved expanding the device list that the hypervisor supports as well as assigning the devices to the task with the proper permissions. Listing 4.5 shows the additions to the device configuration list that the hypervisor uses after initialization, to allow or deny the device to the guest asking to use it.

```
1 {
2  2, DEV_DCFG, 0xFFFFFFFF, "devcfg"
3 },
4 {
5  2, DEV_AXIDMA, 0xFFFFFFFF, "axidma"
6 },
7 {
8   2, DEV_DPR, 0xFFFFFFFF, "dprdev"
9 }
```

Listing 4.5: Configuring the devices for use by the Task.

Analogously to the devices, there was the necessity to configure the required interrupts and ports. Listing 4.6 shows the example of the configuration of the interrupts 61 and 62, which are on the interrupt scope of PL to PS and are used by the partition number 2, which is the service.

```
1 {
2  partition: 2,
3  interrupt: 61,
4  rights: 0x3FFFFFFF,
5  name: "dma_irq"
6 }
7 ,
8 {
9  partition: 2,
10  interrupt: 62,
11  rights: 0x3FFFFFFF,
12  name: "task0_irq"
13 }
```

Listing 4.6: Interrupts Configuration.

Listing 4.7 on the other hand shows the example configuration of a port, in which the partition 1 is able to send messages and grant capabilities to every partition that has capabilities to receive messages on the port.

```
1 {
2  mConfigId : 31,
3  mOwner : 1,
4  mRights : (0x01 << TzPortOperations::SEND) | (0x01 <<
       TzCapabilityOperations::GRANT),
5  mName : "port1"
6 }
```

Listing 4.7: Send and Grant Port configuration for Partition 1.

Another example port configuration is presented in Listing 4.8 in which the partition 1 is able to receive messages and configure the port on runtime.

```
1 {
2  mConfigId : 30,
3  mOwner : 1,
```

```
4  mRights : (0x01 << TzPortOperations::RECV) | (0x01 <<
       TzPortOperations::CONFIG),
5  mName : "port"
6 }
```

Listing 4.8: Receive Port configuration for Partition 1.

One of the last modifications was focused on making guests compile with private configurations and to do so, regions were added to their linker script that contain bitstreams for each algorithm they want as the example on Listing 4.9.

```
1 MEMORY
2 {
3  ps7_ddr_0_S_AXI_BASEADDR : ORIGIN = 0x04000000, LENGTH = 0
       x003FFFFF
4  BITSTREAMS (rw): ORIGIN = 0x04400000, LENGTH = 0x040000
5 }
```

Listing 4.9: Linker script edit for guest partitions.

The last integration stage was to guarantee that the ports and device capabilities were configured correctly in order to allow their owners to use them, and so each partition on start-up get their respective capabilities and configures the ports, devices and interrupts that they use onwards. Listing 4.10 shows the example of the service task configuring the interrupt for the hardware RP.

```
1  if(SetPortInterrupt(st.task0_irq_cap, st.kernel_port_cap ) <
       0) {
2   sw_printk("TASK 0: Failed to configure port cap !\n");
3   return XST_FAILURE;
4  }
5  if(ConfigInterrupt(st.task0_irq_cap, &args) < 0){
6   sw_printk("TASK 0: Failed to configure task0 irq cap!\n");
7   return XST_FAILURE;
8  }
```

Listing 4.10: Configuraring Interrupt on the partition code.

After all the configurations and adjustments were complete the project reached its last stage and from now onwards the verification and evaluation of the implementation will be given.

5. Evaluation

This section presents the evaluation done to the implemented solution towards a TrustZone-assisted hypervisor supporting DPR. The engineering effort, memory footprint, hardware costs and performance obtained on the experimental setup tested are presented and discussed.

5.1 Experimental Setup

The implementation was evaluated on the ZYBO Zynq-7000 SoC, in a single core configuration of the ARM Cortex-A9 running at 650MHz and the hardware logic running at 100MHz. The FPGA can handle running at 250MHz but in this implementation it is limited according to the maximum working frequency of the ICAP interface. The project was developed and synthesized using the Vivado 2018.2 where the IP hardware modules were implemented in Verilog. Vivado HLS 2018.2 was also used to quickly implement hardware modules in order to do a relevant application scenario. For the software development the tool of choice was the XSDK 2018.2, also provided by Xilinx.

The evaluation is focused mainly on the impact on guest performance related to the interaction with the DPR framework, as well as the improvements hardware based accelerators can provide to them. The impact of the modifications necessary to expand the μRTZVisor to support PR mechanisms are also evaluated. Both the hypervisor and partition code were compiled using the Xilinx ARM GNU toolchain.

5.2 Engineering Effort

The engineering effort required for the implementation of the developed code was measured using the Understand software tool, in both software and hardware languages.

Additionally, it allowed to measure the engineer effort associated incurred by each RP added to the system and the modifications necessary to each of the components it interacts with.

5.2.1 Hardware

The hardware developed for the framework allowing for reliable response times and correct assignments to the requests of hardware accelerators represented the most engineering effort to the project. The baseline RP Manager with just one RP on the system, represents an effort of 370 lines of code. From that point, each RP added to the system represents an increase of 28% in lines of code for each RP in the system when compared to the system with only one RP, which is related to the states added to the control unit presented in the implementation chapter. While the increase is considerable, only 30 of the added 105 template lines need slightly changes, mainly connected to asserting the identification of the new RP, with new lines not being required. Each consequent RP represents Figure 5.1 shows the lines of code required up to the limit of 15 RPs.

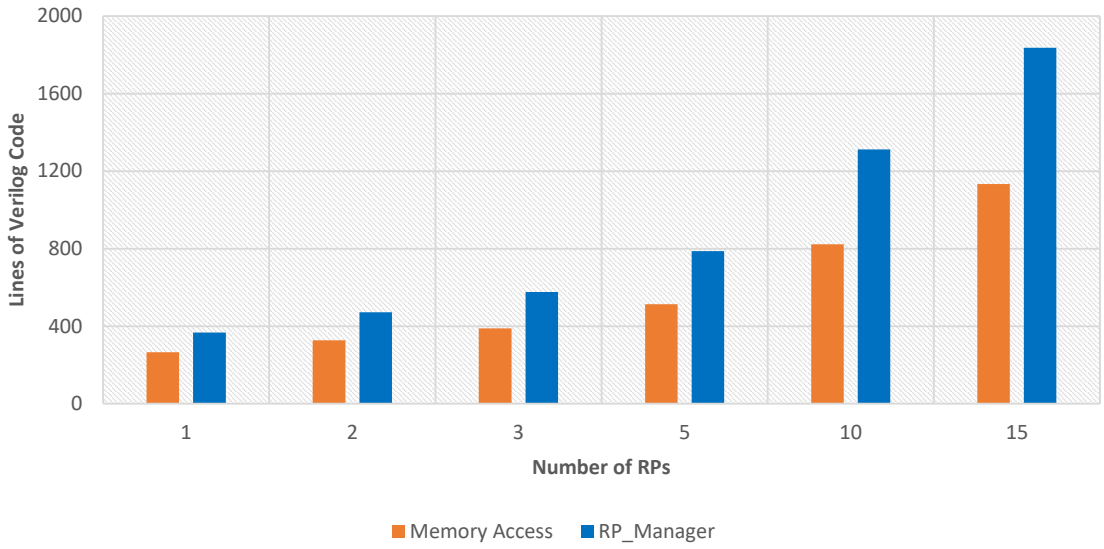


Figure 5.1: Engineering Effort related to the number of RPs on the Hardware.

The baseline Memory Access module with just one RP one the system, represents an effort of 265 lines of code. Each RP added to the system represents an increase of 23% in the lines of code relative to the system with only one RP. In a similar way to the RP Manager, the Memory Access module increase in lines of code is related to the states introduced in the control unit, and also only 22 of the added 62 template lines per RP need modifications.

The RPs are composed of 95 lines of code that are common to every one and are related to the interfaces to the Memory Access and RP_Manager, with engineering effort being only dependent on lines of code of each algorithm implemented as the hardware accelerators.

5.2.2 µRTZVisor Modifications

On the µRTZVisor, the required modifications were related to the integration of the DPR mechanism and the components it requires to work, which represented a 4.9% increase to the TCB of the hypervisor, as shown in 5.1.

µRTZVisor	**Lines of Code**
Native	6693
DPR framework	7022

Table 5.1: Impact on the Hypervisor TCB.

The increase is mostly due to the lines of code related to the necessity of the additional peripheral mapping, namely the AXI DMA used to transfer bitstreams and the RP_Manager which needs to be accessed by the software running on the PS.

It required the introduction of additional lines of assembly code for device tables, mechanisms to handle the request to use the above mentioned hardware modules and system configurations to let the software use them. It also produced modifications to the security configurations of the AXI ports which also represented a significant factor in the verified increment.

Last but not least, the design feature of allowing guests to provide private bitstreams, implied the addition of a hypercall and the respective handler in order to allow the translation of the bitstreams address from the virtual memory to physical addresses to be used by the AXI DMA.

5.2.3 Software Task

Figure 5.2 represents the lines of code necessary depending on the number of RPs implemented on the hardware. For the software task all the developed code accounted to 400 lines of code for a system with only one RP with an additional 100 lines of code to the task for each RP. The increase is mainly due to the necessity of keeping track of the accelerators that can be configured into each RP and the necessity to configure the interrupt assigned to the module.

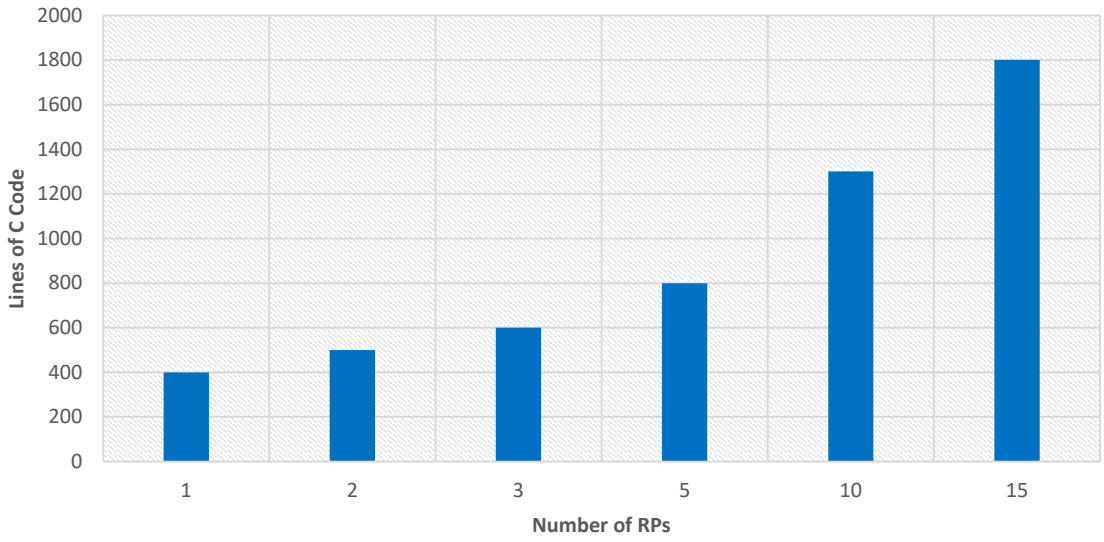


Figure 5.2: Engineering Effort related to the number of RPs on the Software Task.

5.3 Memory Footprint

In order to evaluate the memory footprint of both the native and extended version of the µRTZVisor were compiled using the ARM GNU toolchain.

In Table 5.2 is shown the introduced overhead by each approach on the whole µRTZVisor image, which agglomerates the hypervisor code itself, drivers and libraries. As a consequence of the incurred modifications explained in the previous chapter to the µRTZVisor the modifications are reflected in terms of memory footprint. Nonetheless, the overhead represents only an increase 1.8% in the memory footprint which is acceptable when it encompasses the modifications and adaptations to support DPR.

	Memory Footprint in bytes			
µRTZVisor	.text	.data	.bss	**Total**
Native	143308	40544	105264	289116
DPR framework	143792	43388	107220	294400

Table 5.2: µRTZVisor memory footprint.

5.4 Hardware Costs

To measure the implemented framework hardware resource usage on the ZYBO board, the post implementation hardware utilization report provided by the Vivado tools was analysed. The report provides several information about the usage of each resource present on the matrix of Configurable Logic Blocks (CLBs) and overall impact of the implementation of the synthesised design on the resources used. The ZYBO hardware resources are show in Table 5.3 as a reference point.

Resource	Available
BUFG	32
BRAM	60
FF	35200
LUTRAM	6000
LUT	17600

Table 5.3: Available Resources on the ZYBO SoC.

The implemented design is referent to the minimalistic scenario where 3 RPs were placed on the design and connected to the other modules with all the RPs hosting empty RMs doing only data passing without any logic inside. The test was designed to properly evaluate the resources necessary by the framework and the impact of using the hardware based ICAP and evaluate the resource usage for the increase on the reconfiguration throughput. The results can be seen in Figure 5.3.

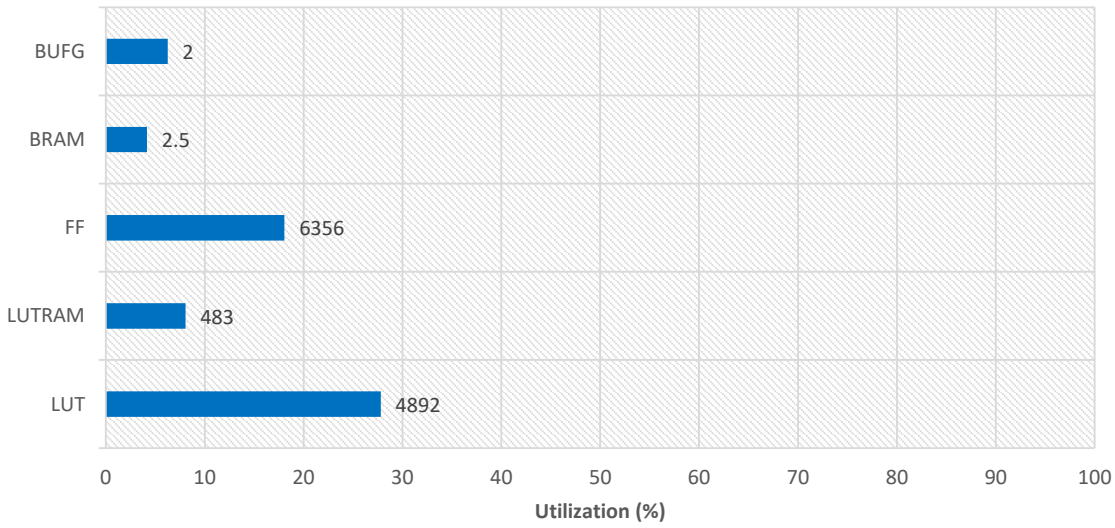


Figure 5.3: Hardware Costs for a minimalistic setup.

Vivado utilization report parameters indicates the number of registers, LUTs, BUFGs, DSPs, FFs, IOs, Block RAM and many other components of the current design required in order to implement it. The more logic is added to the current design, the more resources are usually necessary and the utilization rate of these parameters increases.

5.5 Performance

The performance evaluation process was split into three different phases. First, the reconfiguration mechanisms were tested and compared. Then, the system was evaluated to measure the overhead and latency over the interactions between guests and the service. Finally, the system performance on a intensive test scenario in an environment closer to the expected on real life applications was measured.

5.5.1 Reconfiguration Mechanisms

In order to better understand the impact of each PR mechanism, tested in preliminary stages of the project a test was executed to measure reconfiguration times for different bitstream sizes with every available method. The results presented for this test have been collected in a test application in which the partial bitstream files were stored into the DRAM and the average measured throughput is shown in 5.4 with the PCAP, AXI HWICAP [Xil16b] and PRC [Xil16a] hardware COTS IPs provided by Xilinx, and the custom ICAP accelerated with the AXI_DMA developed for this work.

Mechanism	**Throughput** (MB/s)
AXI HWICAP	14.26
PCAP	126.45
PRC ICAP	256.33
ICAP	389.25

Table 5.4: Average Measured Reconfigurations throughput for each mechanism.

The tests proved that bitstream files describing the same RP even although they are implementing different modules have exactly the same size, which is related to the region and resource type contained in it. The following test was to reconfigure the same RPs with all the ports and measure the reconfiguration time. Figure 5.4 presents the results on the ZYBO board for the mechanism. It

is worth mentioning the AXI HWICAP was purposely left out of the graphic as it had reconfiguration times up to seventy milliseconds, which do not make sense in the real time world. Also the PR controller IP provided by Xilinx, had the same reconfiguration time for different sizes of bitstreams, which represents an odd behaviour when compared to the other methods. This is although due to the algorithm that is implemented in hardware module.

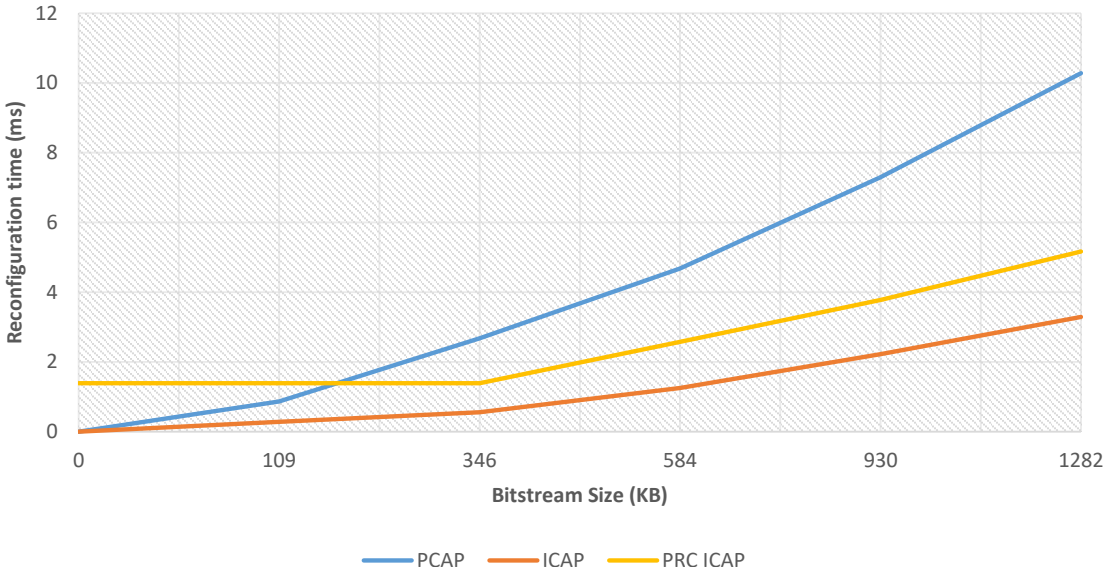


Figure 5.4: Reconfiguration Times for the different Mechanisms.

The measured times validated the dependability of the reconfiguration time on the chosen interface, as given a bitstream size the reconfiguration time only depends on the throughput of the mechanism. This characteristic makes the reconfiguration process predictable and the overhead inserted by PR is known at each reconfiguration.

The test was predominant to choose the standalone custom ICAP as mechanism for the implementation, once the resources used were not extremely high, with the most used resource being 10% of the LUTs. For a reconfigurable region roughly the size of 50% of the FPGA used on the evaluation, the average reconfiguration time was about 2.52ms, which makes the mechanism about four times faster than the more traditional PCAP, while also having the CPU free to process other tasks during the reconfiguration.

5.5.2 RPC Overheads

The last relevant test performed was the one that allowed to measure the impact of the communication mechanism of the hypervisor associated with the

decision time that the software task requires to handle the guest partition requests. The hardware setup implemented on the FPGA fabric was composed of three RPs, a RP Manager and a Memory Access module. On the software, the hypervisor is supervising the secure task alongside one Free-RTOS and two baremetal guests, each guest partition running ten milliseconds of time budget for each execution cycle. All the software was compiled with -03 optimizations in compliance with the evaluation done in [MAC+17]. To properly test the DPR framework, several algorithms were implemented as hardware accelerators and synthesised as RMs to be hosted on a RP and are provided by the software task as reconfigurable accelerators for the guests to request and use. Additionally, one of the baremetal guest partitions was compiled containing a private bitstream in its memory and can request to use it on the hardware.

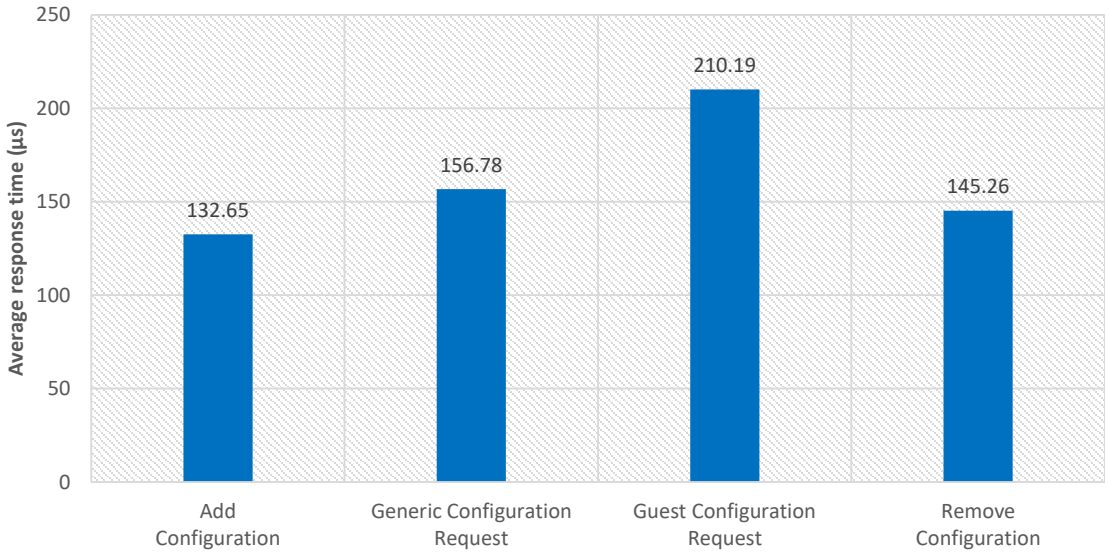
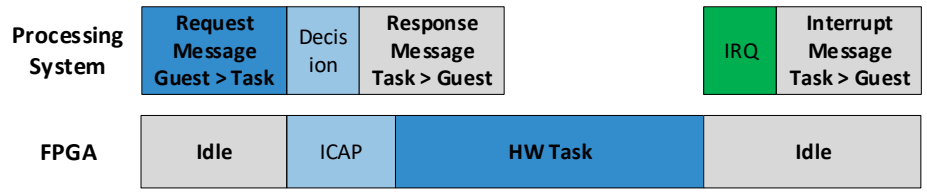
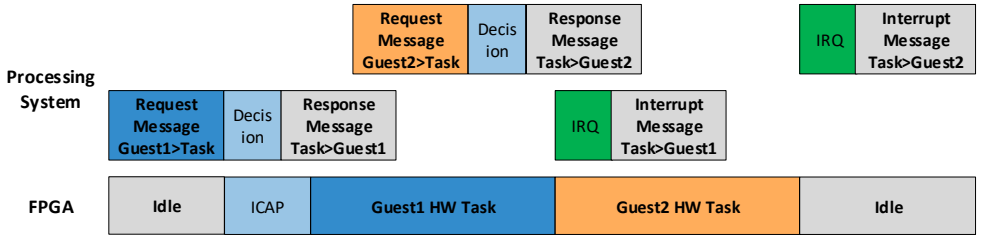


Figure 5.5: Overhead introduced for each Operation.

The measurement to check the overheads linked to the software service decision time, was done in a test designed with the best case scenario, where only the secure task service and one guest partition are running. Figure 5.5 presents the times taken by the software and hardware framework to make a decision in each use case.



(a) Guest request for generic configuration.



(b) Guests 1 and 2 both request the same hardware task configuration sequentially.

Figure 5.6: Execution Flow for Generic Configurations Requests.

In cases as presented in Figure 5.6a, where a guest requests to use a generic configuration, which will require a reconfiguration, the overhead of induced by the task and the communication mechanism represented a average response time of 156 microseconds. An additional test with multiple guest partitions running (Figure 5.6b), showed that the response time is constant despite the number of guests again with the same overheads being witnessed by both requests, revealing the predictability of the framework.

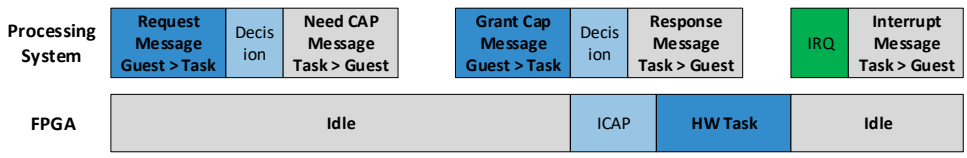


Figure 5.7: Execution Flow for a Private Configuration Request with Reconfiguration.

When tested the use case similar to the one shown in Figure 5.7, where the guest partition makes a request to use a private configuration, the average response was around 210 microseconds. In this use case the average response time is considerably larger than on generic hardware tasks provided by the framework, because the operation introduces two extra messages to the transaction between the task and the guest, which are referent to the necessary capability that the task needs to have, in order to access the memory space of the guest and fetch the bitstream to configure the hardware accelerator.

The response times from the moment that the task receives the message until sends the response to the guest were in the order of the hundreds of microseconds, which represents a fast response mechanism, and is made possible by the fast decision time of the RP Manager, and the low overhead communication mechanism of the hypervisor. The response times are heavily connected to the time that the IPC mechanism of the hypervisor requires to send the message between partitions and having such low overhead mechanism potentiates the framework greatly. It is worth mentioning, that the schedulabity of the partitions involved impacts the induced overhead, and the overhead is subject to partitions being scheduled during the operations.

6. Conclusion

In this modern era, where embedded systems are expanding their presence in our everyday lives, ranging from small IoT devices to systems of high complexity, they all share the same design requirement towards the consolidation of a wide variety of functions into the same hardware platform, while reducing SWaP-C. The rise in the complexity of these systems, represented growing investment in heterogeneous architectures, possessing both CPU and FPGA subsystems, given their great flexibility. These platforms enable fast time-to-market cycles, high adaptability and relatively-low cost, which are all suitable for embedded systems development, while also enabling the extension of the traditional CPU virtualization to FPGAs, which have been further explored for their computing capabilities.

The applicability of co-designed systems that explore the benefits of reconfigurable hardware to ensure the constraints of the embedded domain is used from offloading specific application tasks to hardware, to offloading complete OSes and even hypervisor services. In heterogeneous platforms that have dynamically reconfigurable FPGAs, the DPR technology, which brings the reconfigurable hardware benefits to a new level, has been the focus of recent research. Given its potential for deploying hardware modules on-the-fly, DPR has the ability to bring flexibility one step further, as well as overall improvement, from the multiple hardware accelerators it allows to implement, which otherwise would require much larger logic areas, and in some cases would not even fit on the same platform simultaneously.

This dissertation was focused on designing and implementing a framework for real-time systems that enables real-time applications to exploit the DPR capabilities of the heterogeneous platform, extending the concept of multitasking to the FPGA. The presented framework, was specially developed in order to explore the features of the architecture of μRTZVisor hypervisor, exploiting its communication mechanism and handling of user access to devices. The framework provides a reconfiguration mechanism capable of handling multiple RPs, while providing fast response times, in order to limit the overhead introduced to the normal execution of the operating systems using the framework.

Then a evaluation was done, to evaluate the determinism of the implemented DPR framework, by measuring its latencies and the impact on the normal operation of the OSes using it. The evaluation revealed, a major benefit of the implementation, since the time required to reconfigure a portion of the FPGA is highly predictable, and is dependent on the amount of logic resources involved in the process. For the device used in the evaluation, the Zynq-7000 SoC, the time required to reconfigure an RP of roughly 50% the size of the FPGA is around 2.5 milliseconds. Using the ICAP based reconfiguration process, also enables the reconfiguration to be parallel to the normal execution on the PS. For the specific device used in this study, which has one of the smallest FPGAs in its family, the part of the framework implemented in hardware represented a maximum of 30% of resource usage, which enables the framework to be suitable even for less resourceful platforms, even if with fewer RPs implemented.

Overall, the concept fits well in the concept of a microkernel hypervisor, while still being a relevant feature of μRTZVisor, since the complexity of the kernel remained nearly unmodified, with minimal additions to support the hardware components, and the results appear to be promising.

6.1 Future Work

The proposed framework while being highly flexible in the sense of multiplexing the numerous RPs present on the hardware, presents some limitations that can be improved in future developments. A first step towards improving the system capabilities, would be extending the limited interface between the software and the hardware reconfigurable accelerators, providing more flexibility towards porting hardware accelerators to the implementation. As of the current implementation, the interface is done completely through the software service, and introducing custom ports, that could be directly connected between the guest OSes and the hardware accelerators, could be an interesting starting point. There is also room for improvement in the quantity of processed data by each accelerator, as it is primarily dependent on the AXI HP transfer modes. Dynamically adjustable data size processing could greatly reduce the necessity to request the accelerators multiple times. Additionally, some design automation to enable fast configuration of the framework would also be an interesting improvement.

Furthermore, there is also room for improvements in both the security of the framework and latency that it introduces to the OS using it. As the framework uses the AXI DMA soft core with all the permissions and in secure mode, the hypervisor

mechanisms to control memory access are unable to restring memory access to the mechanism, and as such there is the possibility to jeopardize the hypervisor's memory. Para-virtualizing the AXI DMA could solve this specific problem. In what concerns to the latency, an interesting approach to the framework, would be exploring the potential benefits of using the secondary core present on the platform, dedicated to the DPR framework.

Another objective going forward would be finalizing the implementation of the project presented in [RSTS18], integrating and exploring the benefits of an hardware accelerated IPC and how it can reduce the communication mechanism overhead. Lastly, despite being developed on the Zynq-7000 platform and μRTZVisor in particular, it can be adapted to standalone applications and hypervisors, as well as other heterogeneous platforms enabled with dynamic reconfigurable FPGA present on the market.

References

[AH10] A. Aguiar and F. Hessel. Embedded systems' virtualization: The next challenge? In *Proceedings of 2010 21st IEEE International Symposium on Rapid System Protyping*, pages 1–7, June 2010.

[ARM09] ARM Limited. ARM Security Technology. Building a Secure System using TrustZone Technology ARM. *ARM white paper*, April 2009.

[ARM11] ARM Limited. AMBA AXI and ACE Protocol Specification. October 2011.

[BBCS14] R. Bonamy, S. Bilavarn, D. Chillet, and O. Sentieys. Power consumption models for the use of dynamic and partial reconfiguration. *Microprocessors and Microsystems*, 38(8):860–872, November 2014.

[BBP+16] A. Biondi, A. Balsini, M. Pagani, E. Rossi, M. Marinoni, and G. Buttazzo. A Framework for Supporting Real-Time Applications on Dynamic Reconfigurable FPGAs. In *Proceedings of the 2016 IEEE Real-Time Systems Symposium (RTSS)*, pages 1–12, November 2016.

[BHH+07] J. Becker, M. Hubner, G. Hettich, R. Constapel, J. Eisenmann, and J. Luka. Dynamic and Partial FPGA Exploitation. *Proceedings of the IEEE*, 95(2):438–452, February 2007.

[BSB+14] S. Byma, J. Steffan, H. Bannazadeh, A. Garcia, and P. Chow. FPGAs in the Cloud: Booting Virtualized Hardware Accelerators with OpenStack. In *Proceedings of the 2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 109–116, May 2014.

[Dig16] Digilent Inc. ZYBO FPGA Board Reference Manuals. [Online]. Available: `https://reference.digilentinc.com/_media/zybo:zybo_rm.pdf`, April 2016.

[EH13] K. Elphinstone and G. Heiser. From L3 to seL4 - What Have We Learnt in 20 years of L4 Microkernels? In *Proceedings of the Twenty-Fourth*

ACM Symposium on Operating Systems Principles - SOSP '13, November 2013.

[FLWH10] T. Frenzel, A. Lackorzynski, A. Warg, and H. Härtig. ARM TrustZone as a Virtualization Technique in Embedded Systems. In *Proceedings of Twelfth Real-Time Linux Workshop*, October 2010.

[Gol74] R. Goldberg. Survey of virtual machine research. *Computer*, 7(6):34–45, June 1974.

[GPG+15] T. Gomes, S. Pinto, T. Gomes, A. Tavares, and J. Cabral. Towards an FPGA-based edge device for the Internet of Things, September 2015.

[HD07] S. Hauck and A. DeHon. *Reconfigurable Computing: the Theory and Practice of FPGA-based Computing*. November 2007.

[Hei07] G. Heiser. Virtualization for Embedded Systems. *Open Kernel Labs Technology White Paper*, November 2007.

[Hei08] G. Heiser. The Role of Virtualization in Embedded Systems. In *Proceedings of the 1st workshop on Isolation and integration in embedded systems - IIES '08*, pages 11–16, April 2008.

[Hei11] G. Heiser. Virtualizing Embedded Systems: Why Bother? In *Proceedings of the 48th Design Automation Conference, DAC'11*, pages 901–905. ACM, June 2011.

[HEK+07] G. Heiser, K. Elphinstone, I. Kuz, G. Klein, and S. Petters. Towards trustworthy computing systems: Taking microkernels to the next level. *ACM SIGOPS Operating Systems Review*, 41(4):3–11, July 2007.

[HGNB10] M. Hübner, D. Göhringer, J. Noguera, and J. Becker. Fast dynamic and partial reconfiguration data path with low hardware overhead on Xilinx FPGAs. In *Proceedings of the 2010 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pages 1–8, April 2010.

[HHL+97] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter. The Performance of μ-Kernel-Based Systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, October 1997.

[HN06] C. Hilton and B. Nelson. PNoC: a flexible circuit-switched NoC for FPGA-based systems. *IEE Proceedings - Computers and Digital Techniques*, 153(3):181–188, May 2006.

[HPHS04] M. Hohmuth, M. Peter, H. Härtig, and J. Shapiro. Reducing TCB size

by using untrusted components - small kernels versus virtual-machine monitors. In *Proceedings of the 11th workshop on ACM SIGOPS European workshop*, September 2004.

[HRL+08] H. Härtig, M. Roitzsch, A. Lackorzynski, B. Döbel, and A. Böttcher. L4 – Virtualization and Beyond. *Korean Information Science Society Review*, 2, April 2008.

[JPC+14] A. Jain, K. Pham, J. Cui, S. Fahmy, and D. Maskell. Virtualized Execution and Management of Hardware Tasks on a Hybrid ARM-FPGA Platform. *Journal of Signal Processing Systems*, 77(1-2):61–76, October 2014.

[Kai09] R. Kaiser. Complex embedded systems - a case for virtualization. In *Proceedings of the 2009 Seventh Workshop on Intelligent solutions in Embedded Systems*, pages 135–140, June 2009.

[KK12] D. Kleidermacher and M. Kleidermacher. Embedded Systems Security: Practical Methods for Safe and Secure Software and Systems Development. *Elsevier Inc.*, March 2012.

[KLJ+13] S. Kim, C. Lee, M. Jeon, H. Kwon, H. Lee, and C. Yoo. Secure device access for automotive software. In *Proceedings of the 2013 International Conference on Connected Vehicles and Expo (ICCVE)*, pages 177–181, December 2013.

[LCP+17] P. Lucas, K. Chappuis, M. Paolino, N. Dagieu, , and D. Raho. VOSYSmonitor, a Low Latency Monitor Layer for Mixed-Criticality Systems on ARMv8-A. In *Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS)*, June 2017.

[Lie96] J. Liedtke. Toward real microkernels. *Communications of the ACM*, September 1996.

[LKLJ09] M. Liu, W. Kuehn, Z. Lu, and A. Jantsch. Run-time partial reconfiguration speed investigation and architectural design space exploration. In *Proceedings of the 2009 International Conference on Field Programmable Logic and Applications*, pages 498–502, August 2009.

[LOG+03] T. Lenart, V. Owall, M. Gustafsson, M. Sebesta, and P. Egelberg. Accelerating signal processing algorithms in digital holography using an fpga platform. In *Proceedings of the 2003 IEEE International Conference on Field-Programmable Technology (FPT)*, January 2003.

[LP09] E. Lübbers and M. Platzner. Reconos: Multithreaded programming for

reconfigurable computers. *ACM Transactions on Embedded Computing Systems (TECS)*, 9(1):8–33, October 2009.

[LPFG10] S. Liu, R. Pittman, A. Forin, and J. Gaudiot. On energy efficiency of reconfigurable systems with run-time partial reconfiguration. In *Proceedings of the ASAP 2010 - 21st IEEE International Conference on Application-specific Systems, Architectures and Processors*, pages 265–272, July 2010.

[MAC+17] J. Martins, J. Alves, J. Cabral, A. Tavares, and S. Pinto. μRTZVisor: a Secure and Safe Real-Time Hypervisor. *Electronics*, 6, no. 4:93, October 2017.

[McD08] E. McDonald. Runtime FPGA Partial Reconfiguration. In *Proceedings of the 2008 IEEE Aerospace Conference*, pages 1–7, March 2008.

[PG74] G. Popek and R. Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 7(17):412—42, July 1974.

[PJC+13] K. Pham, A. Jain, J. Cui, S. Fahmy, and D. Maskell. Microkernel hypervisor for a hybrid ARM-FPGA platform. In *Proceedings of the 2013 IEEE 24th International Conference on Application-Specific Systems, Architectures and Processors*, pages 219–226, June 2013.

[POP+14] S. Pinto, D. Oliveira, J. Pereira, N. Cardoso, M. Ekpanyapong, J. Cabral, and A. Tavares. Towards a lightweight embedded virtualization architecture exploiting arm trustzone. In *Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (ETFA)*, pages 1–4, September 2014.

[PPG+17] S. Pinto, J. Pereira, T. Gomes, A. Tavares, and J. Cabral. LTZVisor : TrustZone is the Key. In *Proceedings of the 29th Euromicro Conference on Real-Time Systems, Leibniz International Proceedings in Informatics (LIPIcs)*, volume 76, pages 4:1–4:22, June 2017.

[PRAM17] M. Valdes Pena, J. Rodriguez-Andina, and M. Manic. The internet of things: The role of reconfigurable platforms. *IEEE Industrial Electronics Magazine*, 11(3):6–19, September 2017.

[PS18] S. Pinto and N. Santos. Demystifying Arm TrustZone: A Comprehensive Survey. *ACM Computing Surveys*, preprint, 2018.

[PTM16] S. Pinto, A. Tavares, and S. Montenegro. Hypervisor for Real Time Space Applications. In *Proceedings of the The 4S Symposium*, June

2016.

[RG05] M. Rosenblum and T. Garfinkel. Virtual machine monitors: current technology and future trends. *Computer*, 38(5):39–47, May 2005.

[RSTS18] J. Ribeiro, N. Silva, A. Tavares, and S.Pinto. A TrustZone-assisted Hypervisor Supporting Dynamic Partial Reconfiguration. In *Proceedings of the XIV Jornadas sobre Sistemas Reconfiguráveis*, pages 8 – 11, February 2018.

[SBM+16] J. Shuja, K. Bilal, S. A. Madani, M. Othman, R. Ranjan, P. Balaji, and S. U. Khan. Survey of Techniques and Architectures for Designing Energy Efficient Data Centers. *IEEE Systems Journal*, 10(2):507–519, June 2016.

[SHT13] D. Sangorrín, S. Honda, and H. Takada. Reliable and Efficient Dual-OS Communications for Real-Time Embedded Virtualization. *Information and Media Technologies*, October 2013.

[SN05] J. Smith and R. Nair. Virtual Machines: Versatile Platforms for Systems and Processes. *Elsevier Inc.*, June 2005.

[SRSW14] N. Santos, H. Raj, S. Saroiu, and A. Wolman. Using ARM TrustZone to build a trusted language runtime for mobile applications. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 67–80, March 2014.

[TCL09] D. Thomas, J. Coutinho, and W. Luk. Reconfigurable computing: Productivity and performance. In *Proceedings of the 2009 Conference Record of the Forty-Third Asilomar Conference on Signals, Systems and Computers*, pages 685–689, November 2009.

[THB06] S. Tanenbaum, J. Herder, and H. Bos. Can We Make Operating Systems Reliable and Secure? *Computer*, 39(5):44–51, May 2006.

[VS14] K. Vipin and S.Fahmy. ZyCAP: Efficient Partial Reconfiguration Management on the Xilinx Zynq. *IEEE Embedded Systems Letters*, 6(3):41–44, September 2014.

[Win08] J. Winter. Trusted computing building blocks for embedded linux-based arm trustzone platforms. In *Proceedings of the 3rd ACM workshop on Scalable trusted computing*, pages 21–30, October 2008.

[Xia16] T. Xia. Research on Virtualisation Technology for Real-time Reconfigurable Systems. *Electronics*, December 2016.

[Xil16a] Xilinx Inc. Partial Reconfiguration Controller (v1.0): LogiCORE IP Product Guide (PG193). [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/prc/v1_0/pg193-partial-reconfiguration-controller.pdf, April 2016.

[Xil16b] Xilinx Inc. AXI HWICAP v3.0: LogiCORE IP Product Guide (PG134). [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/axi_hwicap/v3_0/pg134-axi-hwicap.pdf, October 2016.

[Xil17] Xilinx Inc. Vivado Design Suite User Guide: Partial Reconfiguration UG909 (v2017.1). [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_1/ug909-vivado-partial-reconfiguration.pdf, April 2017.

[Xil18a] Xilinx Inc. AXI DMA v7.1: LogiCORE IP Product Guide (PG021). [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/axi_dma/v7_1/pg021_axi_dma.pdf, April 2018.

[Xil18b] Xilinx Inc. Zynq-7000 SoC Technical Reference Manual UG585 (v1.12.2). [Online]. Available: https://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf, July 2018.

[XPN15a] T. Xia, J. Prevotet, and F. Nouvel. An ARM-based Microkernel on Reconfigurable Zynq-7000 Platform. *Mediterranean Telecommunication Journal*, 5(2):109–115, April 2015.

[XPN15b] T. Xia, J. Prevotet, and F. Nouvel. Mini-NOVA: A Lightweight ARM-based Virtualization Microkernel Supporting Dynamic Partial Reconfiguration. In *Proceedings of the 2015 IEEE International Parallel and Distributed Processing Symposium Workshops*, pages 71–80, May 2015.